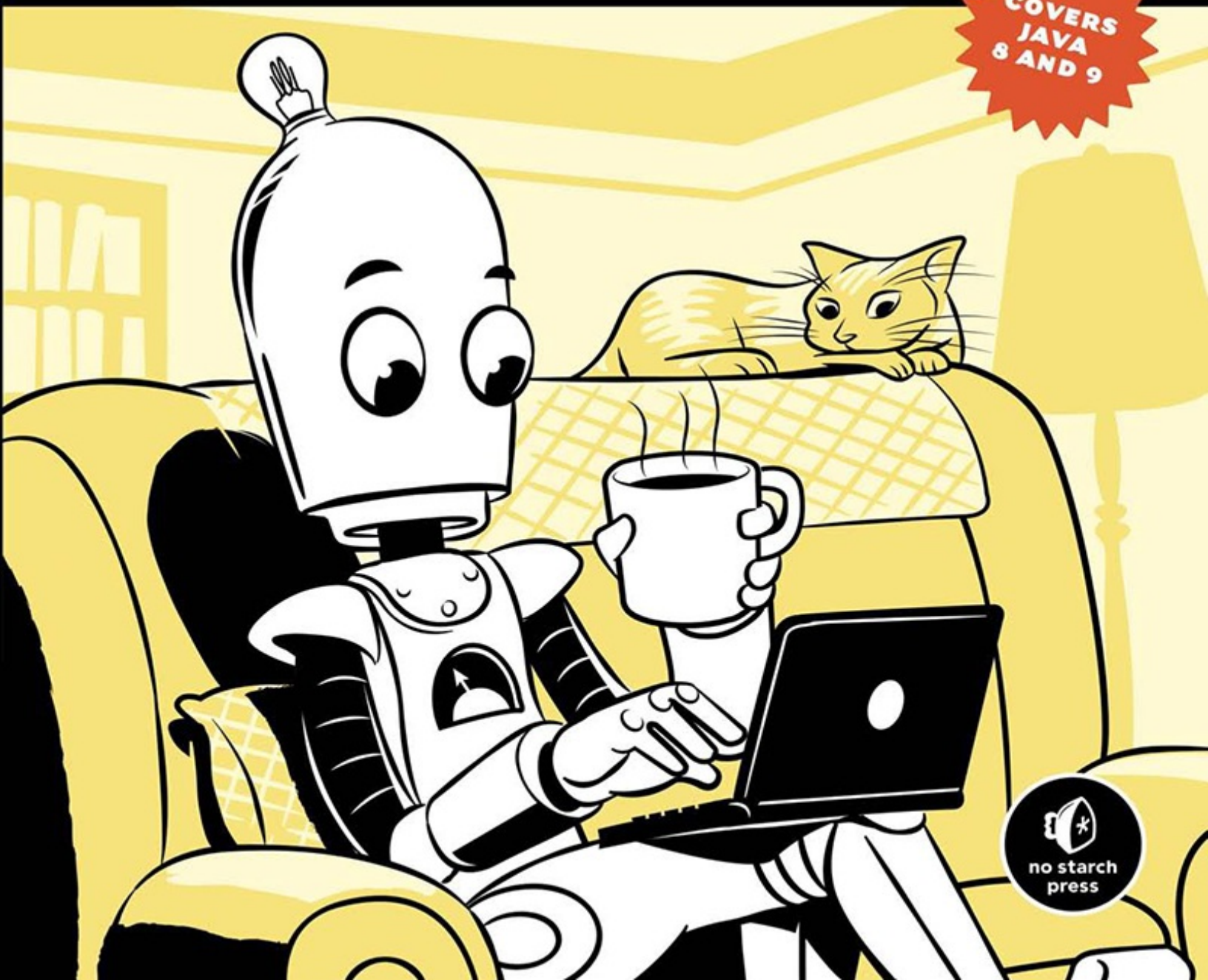
LEARN JAVA THE EASY WAY

A *HANDS-ON* INTRODUCTION TO PROGRAMMING

BRYSON PAYNE

COVERS JAVA 8 AND 9



no starch press

LEARN JAVA THE EASY WAY

LEARN JAVA THE EASY WAY

A Hands-On Introduction to Programming

Bryson Payne

San Francisco

LEARN JAVA THE EASY WAY. Copyright © 2018 by Bryson Payne.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-10: 1-59327-805-5
ISBN-13: 978-1-59327-805-2

Publisher: William Pollock
Production Editor: Riley Hoffman
Cover Illustration: Josh Ellingson
Interior Design: Octopod Studios
Developmental Editors: Jan Cash and Tyler Ortman
Technical Reviewer: Bryan Fagan
Copyeditor: Rachel Monaghan
Compositors: Riley Hoffman and Meg Sneeringer
Proofreader: Paula L. Fleming
Indexer: BIM Creatives, LLC.

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; sales@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Names: Payne, Bryson, author.
Title: Learn Java the easy way : a hands-on introduction to programming / Bryson Payne.
Description: San Francisco : No Starch Press, Inc., [2018]
Identifiers: LCCN 2017022283 (print) | LCCN 2017035701 (ebook) | ISBN 9781593278465 (epub)
   | ISBN 1593278462 (epub) | ISBN 9781593278052 (pbk.) | ISBN 1593278055 (pbk.)
Subjects: LCSH: Java (Computer program language) | Java (Computer program language)
Classification: LCC QA76.73.J38 (ebook) | LCC QA76.73.J38 P3955 2018 (print) | DDC
   005.13/3--dc23
LC record available at https://lccn.loc.gov/2017022283
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch

Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Dr. Bryson Payne is a tenured professor of computer science at the University of North Georgia, where he has taught aspiring coders since 1998. He is a Certified Information Systems Security Professional (CISSP) and Certified Ethical Hacker (CEH), among other industry certifications. He was also the first Department Head of Computer Science at UNG and the founding Director of the Center for Cyber Operations Education, an NSA Center of Academic Excellence in Cyber Defense. He enjoys working with K–12 schools worldwide to promote computer science education.

Featured in *CIO* magazine, *Campus Technology*, and the *Wall Street Journal*, Dr. Payne is also the best-selling author of *Teach Your Kids to Code: A Parent-Friendly Guide to Python Programming* (No Starch Press, 2015), now published in five languages (Korean, Portuguese, Russian, Chinese, and English). He's also a popular online instructor of several courses, including the highly rated Udemy course *Real-World Ethical Hacking* (2017). His courses reach more than 20,000 enrolled students from more than 150 countries. He holds a PhD in computer science from Georgia State University and has published articles in scholarly and trade journals, in addition to speaking regularly at regional and national conferences on computer science and cybersecurity education.

Dr. Payne lives north of Atlanta, Georgia, with his lovely wife, Bev; two sons, Alex and Max; and three cats, Leo, Rocky, and Pixel.

About the Technical Reviewer

Bryan Fagan is a middle school engineering and technology teacher. He lives in Dahlonega, Georgia, where he teaches, enjoys the slow-rolling Appalachian mountains with his wife and daughter, and trains in Brazilian jiujitsu. He is the founder of Make Stuff, an after-school makerspace for middle school students.

To Mom, for always believing in me

BRIEF CONTENTS

Acknowledgments

Introduction

Chapter 1: Getting Started

Chapter 2: Build a Hi-Lo Guessing Game App!

Chapter 3: Creating a GUI for Our Guessing Game

Chapter 4: Creating Your First Android App

Chapter 5: Polishing Your App by Adding Menus and Preferences

Chapter 6: Deciphering Secret Messages

Chapter 7: Creating Advanced GUIs and Sharing Your App

Chapter 8: Make Secret Messages a Phone App to Share with Friends!

Chapter 9: Paint Colorful Bubbles with Your Mouse!

Chapter 10: Adding Animation and Collision Detection with Timers

Chapter 11: Making BubbleDraw a Multitouch Android App

Appendix: Debugging and Avoiding Common Errors in Java

Index

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

INTRODUCTION
Why Should I Learn to Code?
Why Should I Learn Java?
What's in This Book
What Tools Do I Need?
Online Resources
Start Now!

1
GETTING STARTED
Java on Windows, macOS, and Linux
Installing Java 8 and 9 for Developers
Installing the Eclipse IDE for Java Developers
Setting Up Eclipse
    Installing the WindowBuilder Editor
    Customizing Eclipse's Look and Feel
Installing Android Studio for Mobile App Development
Getting to Know Java with JShell
    Running JShell
    Working with Java Expressions in JShell
    Declaring Java Variables in JShell
    Printing Output in Java
    JShell Commands
What You Learned

2
BUILD A HI-LO GUESSING GAME APP!
Planning the Game Step-by-Step

Creating a New Java Project
Creating the HiLo Class
    Generating a Random Number
    Getting User Input from the Keyboard
    Making the Program Print Output
Loops: Ask, Check, Repeat
    if Statements: Testing for the Right Conditions
    Adding a Play Again Loop
Testing the Game
What You Learned
Programming Challenges
    #1: Expanding Your Range
    #2: Counting Tries
    #3: Playing MadLibs

3
CREATING A GUI FOR OUR GUESSING GAME
Practicing with JShell
    Creating a GUI in Four Lines of Code
    Creating an Interactive GUI in 10 Lines of Code!
Setting Up the GUI App in Eclipse
GUI Design with Eclipse's WindowBuilder Editor
Designing the User Interface
    Setting GUI Properties in the Properties Pane
    Customizing GUI Components in the Palette Pane
    Aligning GUI Elements
    Naming GUI Components for Coding
    Connecting the GUI to Your Java Code
Adding a Method to Check the Player's Guess
    Getting Text from a JTextField
    Converting Strings to Numbers
Starting a New Game
Listening for User Events: Click to Guess!
Setting Up the GUI Window

Time to Play!
Adding a Play Again Feature
Improving the UX
    Allowing Users to Press Enter to Guess
    Automatically Removing Old Guesses
Handling Bad User Input
What You Learned
Programming Challenges
    #1: Showing Users How Many Tries They Took
    #2: Showing and Hiding a Play Again Button
    #3: Creating a GUI MadLib

4
CREATING YOUR FIRST ANDROID APP
Starting a New Android Studio App Project
Building the GUI Layout in Design View
Naming GUI Components in Android Studio
Connecting the GUI to Java in Android Studio
Adding Methods to Check the Guess and Begin a New Game
Handling Events in Android
Running the App on the Android Emulator
Running the App on a Real Android Device
    Preparing Your Device
    Connecting Your Device
    Running the App on Your Device
Improving the UX
    Centering the User's Guess in the Text Field
    Adding a Listener for the Enter Key
    Adding One More Finishing Touch
What You Learned
Programming Challenges
    #1: "Toast"-ing to the Number of Tries
    #2: Adding Visual Appeal
    #3: Creating a MadLibs Mobile App

5
POLISHING YOUR APP BY ADDING MENUS AND PREFERENCES
Adding an Options Menu in Android
Adding Items to the Menu's XML File
Displaying the Options Menu
Responding to User Selections
Creating an Alert Dialog Pop-up for the About Screen
Changing the Guessing Range
Adding a Variable for the Range
Using the range Variable
Building the Dialog to Allow the User to Select the Range
Storing User Preferences and Game Stats
Storing and Retrieving the User's Preferred Range
Storing the Number of Games Won
What You Learned
Programming Challenges
#1: You Win Some, You Lose Some
#2: Ratio of Wins to Losses

6
DECIPHERING SECRET MESSAGES
The Caesar Cipher
Setting Up the Secret Messages App
Creating the Secret Messages Project in Eclipse
Beginning to Code SecretMessages.java
Messing with Strings
Characters and Values in Java
Encoding Just the Letters
Closing the Scanner
Adding a Custom Key Value
Encoding Digits
Running Command Line Apps Without Eclipse
Finding Your Workspace Folders
Opening a Command Line Window

What You Learned
Programming Challenges
#1: Looping the Loop
#2: Reversing and Encoding
#3: Safely Handling Keys with try and catch

7
CREATING ADVANCED GUIs AND SHARING YOUR APP
Setting Up the Secret Messages GUI App Project
Designing the GUI and Naming Components
Coding the Secret Messages GUI App
Creating the encode() Method
Writing the Event Handler for the Encode/Decode Button
Handling Bad Input and User Errors
Building the main() Method and Running the App
Improving the GUI
Setting Line Wrap and Word Wrap
Handling Bad Input and User Errors: Part 2
Adding a Slider to the Secret Messages GUI
Code Cracking with the Slider
Bonus: Sharing Your App as a Runnable JAR File
What You Learned
Programming Challenges
#1: Movin' On Up!
#2: Scrolling Away!
#3: Changing the Text to Change the Slider

8
MAKE SECRET MESSAGES A PHONE APP TO SHARE WITH FRIENDS!
Setting Up the Mobile GUI
Designing the Mobile GUI
Wiring the GUI to the Java Code
Connecting the Encode Button to the encode() Method

Testing the App
Working with the SeekBar
Running the App on the Emulator and on an Android Device
Bonus: Customizing the Floating Action Button
Receiving Secret Messages from Other Apps
What You Learned
Programming Challenges
#1: Creating a Move Up Button
#2: Changing the SeekBar's Progress

9
PAINT COLORFUL BUBBLES WITH YOUR MOUSE!
Creating the BubbleDraw Project Files
Building the BubbleDraw Frame
Creating a Class for Bubbles
Defining a Bubble
Designing a Bubble's Methods
Storing Bubbles in an ArrayList
Adding a Constructor to the BubblePanel Class
Adding a Method to Draw on the Screen
Testing the BubblePanel Class
Handling Mouse Events from the User
Creating a Reusable Event Listener
Handling Clicks and Drags
Bonus: Handling MouseWheel Events
What You Learned
Programming Challenges
#1: No Bubble Too Small
#2: PixelDraw!

10
ADDING ANIMATION AND COLLISION DETECTION WITH TIMERS
Copying the BubbleDraw Java Project to Create BubbleDrawGUI

Renaming the Main Class and Java File
Adding Transparency
Adding Animation: Bubbles Rising!
Adding a Timer
Setting the Timer
Preparing the Animation
Starting the Timer
Forever Blowing Bubbles: Adding Random Speed and Direction
Building a GUI for Our Animated Drawing App
Setting Up the GUI Panel and Buttons
Coding the Clear and Pause/Start Buttons
Bouncing off the Walls with Collision Detection
A Soft Bounce
A Hard Bounce
Adding a Slider to Control the Animation Speed
Customizing the Slider
Implementing the Slider Event Handler
What You Learned
Programming Challenges
#1: No Bubble Left Behind
#2: Flexi-Draw!
#3: PixelDraw 2.0

11
MAKING BUBBLEDRAW A MULTITOUCH ANDROID APP
Setting Up the BubbleDraw Project
Creating the BubbleView Constructor
Adding the Animation Variables
Creating the BubbleView() Constructor
Preparing the Layout to Use BubbleView
Modifying the Bubble Class
Drawing in Android with the onDraw() Method
Testing BubbleDraw with 100 Bubbles
Adding testBubbles()

Fixing the OnTouchListener Error
Running the BubbleDraw App
Using Threaded Animation and Multitasking in Java
Using Touch to Draw with Your Finger
Using Multitouch to Draw with 10 Fingers at a Time!
Testing Multitouch Events on an Android Device
Changing the App Launcher Icon
Creating a Custom App Icon
Adding the Custom Icon to Your App
Displaying Your New Icon
Changing the App Name
What You Learned
Programming Challenges
#1: Combining One-Finger and Multitouch Events, v1.0
#2: Combining One-Finger and Multitouch Events, v2.0

APPENDIX
DEBUGGING AND AVOIDING COMMON ERRORS IN JAVA
Spelling and Case
Correcting Typos in Eclipse
Correcting Typos in Android Studio
Avoiding Other Common Spelling Errors
Comparison Trouble
Grouping Symbols
Quick Fixes in Eclipse
Code Completion in Android Studio
Summary

INDEX

ACKNOWLEDGMENTS

This book would not have been possible without the exceptional support of the No Starch Press team. Thanks especially to Bill Pollock, Tyler Ortman, Riley Hoffman, Jan Cash, Serena Yang, Amanda Hariri, and Julia Borden for their tireless editing, review, and marketing, and for the innumerable ways they helped me improve this book from my original manuscript.

Thanks to Bryan Fagan for his awesome work as a technical reviewer.

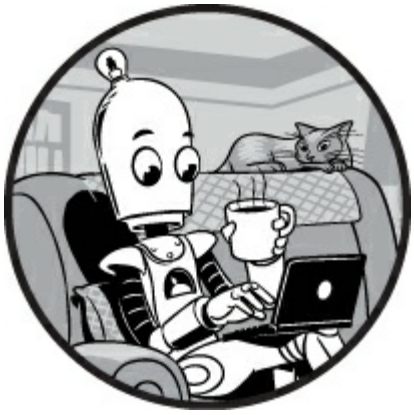
Thanks to all of my current and former students who inspire me to continue to create interesting, engaging content—especially Shah and Susan Rahman, Justin and Diane Turner, Jacob Elliott, Brian Murray, Aaron Walker, Robert Brown, Trent Deal, Seth Park, Simon Singh, Andrew Miller, the Ctrl-Alt-Del robotics team, Jackson Grant, Quintin Kerns, Grace and Jack Halley, Matthew Harpur, and David Knight. Thanks also to the students from 150 countries enrolled in my online courses, especially Hayden Redd and Bob Watson.

Thanks to my inspiring colleagues and friends who continually motivate me to be my best: Markus Hitz, Chuck Robertson, Irene Kokkala, Tamirat Abegaz, Antonio Sanz Montemayor, Don Watkins, Eddie Mienie, Rose Procter, Ron Larson, Victor Parker, Keith Antonia, Billy Wells, Jim Goldy, James Daniel, and Craig Gentry.

Thanks to my father-in-law, Norman Petty, whose passion for technology is surpassed only by his love for family, and my stepdad, Dale Carver, who always has time for my 3D-printed gadgets. Thanks most of all to my beautiful wife, Bev, and my amazing sons, Alex and Max, for their unending patience while I wrote two books in three years.

Special thanks to Kalen Cole. At eleven years old, Kalen embraced my first book, *Teach Your Kids to Code*, learning to code at his own pace. He even sent me his first creative design using lines of code he'd written himself. Way to go, Kalen! You are an inspiration to every kid who's just beginning to discover who they are and what they want to become.

INTRODUCTION



Java is used on billions of devices worldwide. From mobile apps to desktop software, Java powers the largest enterprises and the smallest personal devices. Students, IT professionals, and anyone considering a career in programming will find that they need to learn Java. As a computer science professor who has taught Java for almost 20 years, I wrote this book to help you learn Java the way I learned to code: with hands-on projects. I've found that students learn best by building real apps and games that are interesting, engaging, and worth sharing. In this book, you'll make a simple Hi-Lo guessing game, a Secret Messages app to exchange messages with friends, and an interactive drawing app called BubbleDraw. No prior programming experience is required, but if you have learned other languages, you'll also pick up Java faster using this hands-on approach.

Why Should I Learn to Code?

First, jobs. According to the US Bureau of Labor Statistics, seven of the top ten fastest-growing, highest-paying job fields are in computing. Programmers are in demand worldwide, and millions more will be needed over the next several years. No matter where you are, as long as you have an internet connection, you can make money as a programmer.

But high income and job security aren't the only reasons to learn to code. Coding is problem solving, and the world needs more problem solvers. You can write apps that connect people and help them work. You can enable new forms

of commerce and even create entirely new markets. You can break down barriers, help an individual or a community or a whole continent, and create opportunities that hadn't existed before. Thanks to the reach of the internet and smartphones, you can write an app and share it with billions of people.

Drew Houston, founder of Dropbox, says coding is "the closest thing we have to a superpower," and Gabe Newell, cofounder of video gaming's Valve Corporation, says knowing how to code makes you "look like you have magic powers compared to everybody else." Computers are all around us—in every device, every system, and every network in our daily lives—and code is what makes all those computers work. Learn to code, and you learn to thrive in a high-tech future.

Why Should I Learn Java?

Java is considered by many to be the #1 programming language, for several good reasons. First, it runs on virtually every type of device imaginable, from desktops and laptops to smart TVs. The same Java code will run on Windows, macOS, or Linux.

Second, Java is used by companies to run some of the biggest enterprise applications. Java is an object-oriented programming language, designed to allow software engineers to develop massive applications in fields ranging from manufacturing to sales and human resources to accounting.

Third, Java is one of the most popular languages in colleges and universities worldwide, so it brings you up to speed with fellow coders right away.

Whether you're coding as a hobby, as a side business, or as your full-time job, Java makes it free and easy to get started. Java is a great first language or a terrific *next* language if you've coded before.

What's in This Book

Here's a quick overview of what you'll find in each chapter.

In **Chapter 1**, you'll be guided through installing and setting up Java, Eclipse, and Android Studio. You'll also write your first Java commands using the JShell interactive shell.

In **Chapter 2**, you'll write your first application—the Hi-Lo guessing game, a text-based, command line program that has a player guess a random number between 1 and 100. In **Chapter 3**, you'll upgrade the guessing game into a windowed, desktop application, complete with a graphical user interface (GUI) featuring labels, a text field, and a clickable button.

In **Chapter 4**, you'll build your first Android mobile app, reusing much of the Hi-Lo guessing game code from the previous two chapters. Once you have the mobile app completed, in **Chapter 5**, you'll add several finishing touches to the guessing game, including a settings menu and a high scores feature.

In **Chapter 6**, you'll start a new program—the Secret Messages app, a text-based program that scrambles messages using a Caesar cipher. In **Chapter 7**, you'll give the app a GUI upgrade by adding a slider bar tool that will allow you to crack Caesar ciphers quickly. Then, in **Chapter 8**, you'll make a mobile Secret Messages app with sharing functionality, which will allow users to share secret messages back and forth by email, text message, or social media with the touch of a button.

In **Chapter 9**, you'll begin creating the most visual application in the book—the BubbleDraw app, which allows the user to draw colorful bubbles on the screen. In **Chapter 10**, you'll add animation to make the bubbles float and bounce around the screen as they're drawn, and in **Chapter 11** you'll add multitouch functionality to draw bubbles in several places at once. By the end of this chapter, you'll have a professional-looking app that you'll want to share with all your friends!

Finally, in the **appendix**, you'll find tips for debugging and avoiding common errors as you write Java programs in Eclipse and Android Studio.

What Tools Do I Need?

The programming tools you'll learn to use in this book, Eclipse and Android Studio, are two of the most popular Java development tools used in industry, so once you finish this book, you'll be ready to code real apps right away. Best of all, they're free to download and use!

To get started, all you need is internet access and a regular desktop or laptop computer running Windows, macOS, or Linux. You don't even need an Android phone or tablet to build mobile applications, because Android Studio comes with

a free Android device emulator that you can use to test your apps. Of course, if you have an Android device, you'll be able to run your mobile apps directly on your phone or tablet.

Online Resources

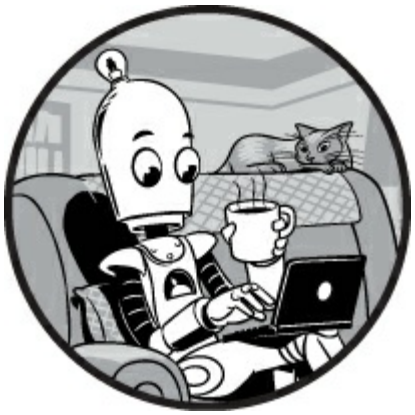
If you want extra help, you can download all the source code for the apps free on the book's official website at *https://www.nostarch.com/learnjava/*. And, if you'd like step-by-step videos in a one-on-one tutorial style, the online course at *http://www.udemy.com/java-the-easy-way/* will guide you through every example and every line of code. Use the coupon code BOOKHALFOFF to save 50 percent on the course. You can ask questions or message me directly in the online course anytime.

Start Now!

You've got nothing to lose and everything to gain by learning to code in Java. Start today. Coding is the key to a new world of possibilities, and learning Java could be your first step into a new career and a new future. Whether young or old, my university students have one thing in common: they took that first step. They wrote their first line of code and their first program, then learned and grew from there. You can do the same thing.

As the Chinese proverb says, "The best time to plant a tree was 20 years ago. The second best time is now." Whether you're still in school or considering embarking on a second career, the time to learn to code is now.

1
GETTING STARTED



In this chapter, you'll get started by installing Java, Eclipse, and Android Studio onto your computer, and then you'll get familiar with some basic Java programming by testing out some commands in the interactive command line shell, JShell.

Java is a powerful, multiplatform programming language that's free to download and install on Windows, macOS, and Linux. Eclipse, an industry-standard *integrated development environment (IDE)*, is a toolkit for building Java apps quickly and easily. And Android Studio is the development environment for Android mobile apps in Java. It gives you the power to program your own mobile games and apps for phones, tablets, and more.

Java on Windows, macOS, and Linux

One of the great things about Java is that the Java programs you write can run on *any* other computer that has the Java Virtual Machine (JVM) software (sometimes called the Java Runtime Environment, or JRE) installed. JVM technology allows you to write a program once and run the same code on any operating system (Windows, macOS, Linux, Android, and more).

This sounds like an obvious idea, but most other programming languages require you either to write different code for Windows, macOS, Linux, and

smartphones or to compile a separate version for each operating system. The JVM is a runtime environment that makes this unnecessary.

Figure 1-1 shows the same graphical Java app running on Windows, macOS, and Ubuntu Linux.

Dr. Payne's Secret Message App
N ymnsp N qtaj htinsl ns Ofaf!
Key: -5
ENCODE/DECODE
-13 0 13
I think I love coding in Java!
Dr. Payne's Secret Message App
N ymnsp N qtaj htinsl ns Ofaf!
Key: -5
ENCODE/DECODE
-13 0 13
I think I love coding in Java!
Dr. Payne's Secret Message App
N ymnsp N qtaj htinsl ns Ofaf!
Key: -5
ENCODE/DECODE
-13 0 13
I think I love coding in Java!

Figure 1-1: The same Java code running on three different operating systems—Windows, macOS, and Linux!

This simple but powerful concept is one of the reasons Java has been adopted by individuals and businesses worldwide.

Installing Java 8 and 9 for Developers

The Java Development Kit (JDK) is the version of Java for developers or programmers—like you! The JDK gives you the power to write and compile your own Java apps that you can share with your friends, deploy in a business, or run anywhere on almost any device.

We're going to install both versions 8 and 9 of the JDK. That way, we can gain the benefits of version 8's widespread use while also accessing version 9's newest features.

To install JDK 8:

1. Go to *http://jdk.java.net/8/*.
2. Click the **Accept License Agreement** radio button.
3. Find the link for your operating system in the list of downloads under JDK, and click it. If your operating system is Windows or Linux, choose the 64-bit version.
4. Open the JDK file from your computer's *Downloads* directory and install the

JDK.

To install JDK 9, go to *http://jdk.java.net/9/* and then repeat steps 2–4 from your JDK 8 installation.

NOTE

For more detailed, step-by-step instructions in video form, you can preview the online course associated with this book for free at http://www.udemy.com/java-the-easy-way/.

That's all it takes to get your computer ready to compile and run Java programs from the text-based command line or terminal, but we also want to take advantage of Java's ability to create *graphical user interface (GUI)* applications like those in Figure 1-1. To tap into Java's GUI awesomeness, next we'll install an integrated development environment called Eclipse.

Installing the Eclipse IDE for Java Developers

Eclipse is one of the most popular IDEs for programming in Java. It's also open source, which means it's free for both personal and business use, and there's a thriving open source community continually improving and supporting Eclipse. There are a number of alternative IDEs for Java, but Eclipse is famous for its ease of use in developing Java apps. It's also pretty easy to install.

To install Eclipse on your computer, go to *http://www.eclipse.org/downloads/*, download the installer for your operating system (as shown in Figure 1-2), and then run it. The current version as of this writing is Eclipse Oxygen.

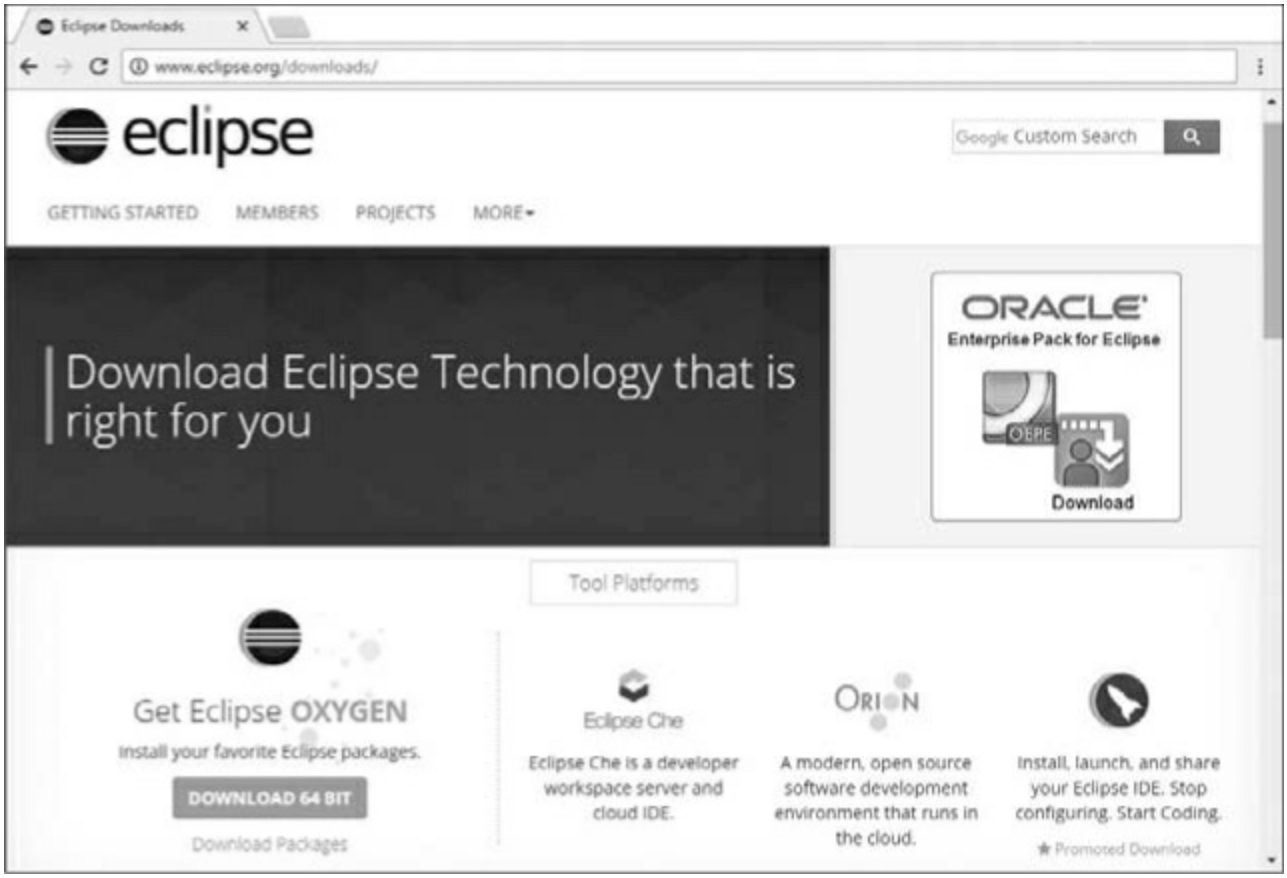


Figure 1-2: Download the Eclipse IDE installer for your operating system.

You'll see a menu like the one in Figure 1-3. Choose **Eclipse IDE for Java Developers** and click **Install**. Be careful to choose the Eclipse IDE for Java Developers option and not one of the other options (the Java EE, or Enterprise Edition, doesn't have some of the features we'll use in this book).

Installation may take several minutes to complete.

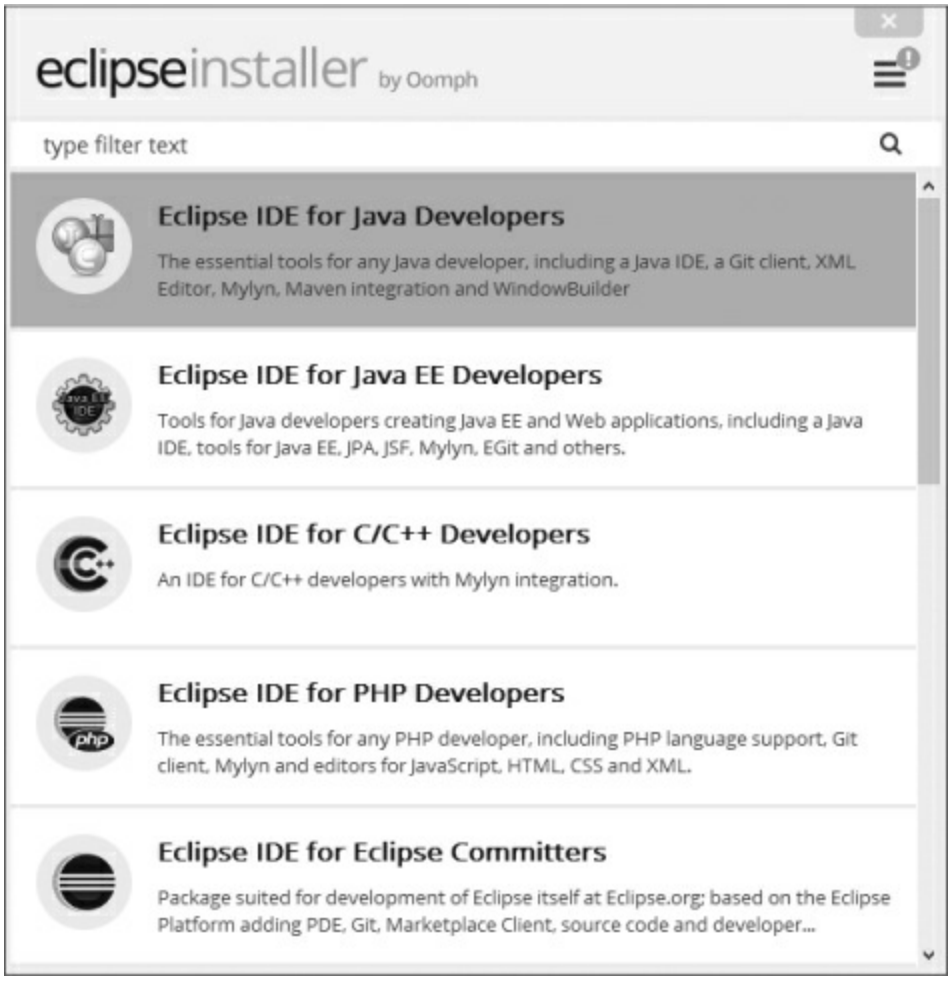

eclipseinstaller by Oomph
type filter text
Eclipse IDE for Java Developers
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven integration and WindowBuilder
Eclipse IDE for Java EE Developers
Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.
Eclipse IDE for C/C++ Developers
An IDE for C/C++ developers with Mylyn integration.
Eclipse IDE for PHP Developers
The essential tools for any PHP developer, including PHP language support, Git client, Mylyn and editors for JavaScript, HTML, CSS and XML.
Eclipse IDE for Eclipse Committers
Package suited for development of Eclipse itself at Eclipse.org; based on the Eclipse Platform adding PDE, Git, Marketplace Client, source code and developer...

*Figure 1-3: Choose **Eclipse IDE for Java Developers** from the Eclipse installer menu.*

Setting Up Eclipse

Let's get our Eclipse installation configured to look and feel like the professional development environment it is, complete with the WindowBuilder Editor, a coder-friendly color scheme, and easy-to-read fonts.

Start Eclipse by clicking the Eclipse icon. When Eclipse starts, it will usually ask you where you want to store your *workspace*, which holds all of your Java projects, as shown in Figure 1-4. You can use the default location (on Windows it's *C:\Users\<YourUserName>\eclipse-workspace*, on macOS it's */Users/<YourUserName>/Documents/eclipse-workspace/*, and on Linux it's */home/<YourUserName>/eclipse-workspace/*), or you can choose a custom location for your Java workspace folder.

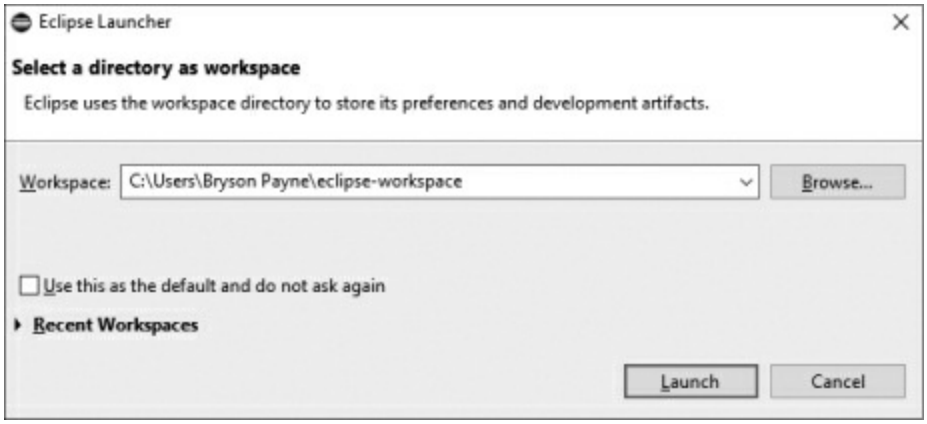


Figure 1-4: Eclipse begins by asking where you would like to store your Java projects.

If you don't have another preference, use the default *eclipse-workspace* location. No matter what, remember the location of this folder because this is where all your Java projects will be stored. If you check the box labeled "Use this as the default and do not ask again," the Eclipse Launcher window won't pop up every time you start Eclipse. If you plan to use multiple workspaces, you might want to leave the checkbox unchecked so that you can switch between workspaces easily when opening Eclipse.

MODIFYING ECLIPSE TO RUN WITH JAVA 9

If Eclipse fails to launch, there's one more change you'll need to make. As of this writing, Java 9 is still brand-new, and while Eclipse versions Oxygen and later support Java 9, some releases require an addition to the *eclipse.ini* configuration file to run with Java 9 installed. To make this change, follow these steps:

1. Find your Eclipse installation folder:

   - On Windows, you can right-click the Eclipse shortcut and choose **Open file location**. The *eclipse.ini* file will be in the same folder as the *eclipse.exe* program file.

   - On macOS, find the Eclipse app in your *Applications* folder in Finder, CONTROL-click the Eclipse app icon, and choose **Show Package Contents**. Open the *Contents* folder, then the *Eclipse* folder, and you will see *eclipse.ini* in the list of files.

- On Linux, go to your *home* folder and open *eclipse/java-oxygen/eclipse* to find *eclipse.ini*.

2. Right-click or CONTROL-click the *eclipse.ini* file. Choose **Open With . . .** and then select Notepad, TextEdit, or another text editor of your choice.
3. Add the following line to the bottom of the *eclipse.ini* file:

```
--add-modules=ALL-SYSTEM
```

4. Save *eclipse.ini* and then reopen Eclipse. Eclipse should launch correctly from now on.

The first time you run Eclipse, you'll see a welcome screen. Depending on your version of Eclipse, this screen may include some useful sample projects and tutorials, or it might be simpler. Feel free to click around and explore a bit if you'd like, and when you're ready to move ahead, close the Welcome tab by clicking the small *X* at the top.

Installing the WindowBuilder Editor

The most important upgrade we'll make to Eclipse is to install the WindowBuilder Editor, which allows us to build windowed applications by dragging and dropping GUI elements like buttons, labels, and text fields onto a graphical preview of the app.

Some versions of Eclipse have the WindowBuilder Editor already installed, but we'll go over the steps to install or update it to make sure you're ready to build GUI applications starting in Chapter 3.

First, go to *http://www.eclipse.org/windowbuilder/* and click **Download**. On the download page, find the WindowBuilder version that matches your version of Eclipse (for Eclipse Oxygen, that's version 4.7), right-click or CONTROL-click the corresponding link, and copy the link address, as shown in Figure 1-5.

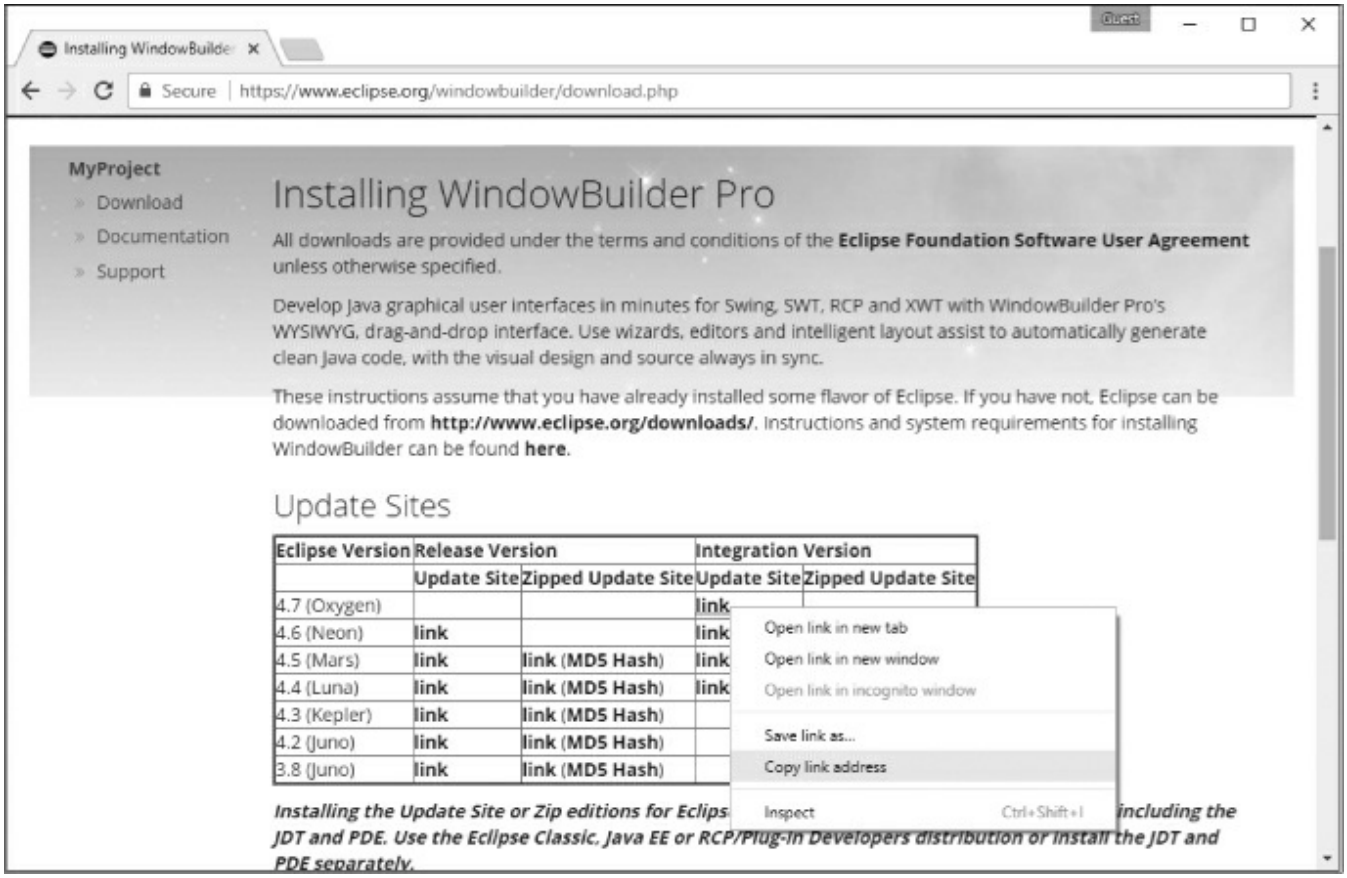

Installing WindowBuilder
Secure | https://www.eclipse.org/windowbuilder/download.php
MyProject
Download
Documentation
Support
Installing WindowBuilder Pro
All downloads are provided under the terms and conditions of the Eclipse Foundation Software User Agreement unless otherwise specified.
Develop Java graphical user interfaces in minutes for Swing, SWT, RCP and XWT with WindowBuilder Pro's WYSIWYG, drag-and-drop interface. Use wizards, editors and intelligent layout assist to automatically generate clean Java code, with the visual design and source always in sync.
These instructions assume that you have already installed some flavor of Eclipse. If you have not, Eclipse can be downloaded from http://www.eclipse.org/downloads/. Instructions and system requirements for installing WindowBuilder can be found here.
Update Sites
Eclipse Version
Release Version
Integration Version
Update Site
Zipped Update Site
4.7 (Oxygen)
4.6 (Neon)
4.5 (Mars)
4.4 (Luna)
4.3 (Kepler)
4.2 (Juno)
3.8 (Juno)
link
link (MD5 Hash)
Open link in new tab
Open link in new window
Open link in incognito window
Save link as...
Copy link address
Inspect
Ctrl+Shift+I
Installing the Update Site or Zip editions for Eclips
including the JDT and PDE. Use the Eclipse Classic, Java EE or RCP/Plug-in Developers distribution or install the JDT and PDE separately.

Figure 1-5: Find the latest WindowBuilder Editor download link and copy the link address.

Next, come back to Eclipse and select **Help ▸ Install New Software**. In the **Work with:** text box, paste the URL of the WindowBuilder Editor (for Eclipse Oxygen, that's *http://download.eclipse.org/windowbuilder/WB/integration/4.7/*), click the **Add...** button, and enter `WB` in the Name field in the pop-up, as shown in Figure 1-6.

Click **OK**, and when the WindowBuilder checkbox appears in the install window, click **Select All** to install all of the needed WindowBuilder components. Click **Next** until you are prompted to accept the license agreement, and finally click **Finish**.

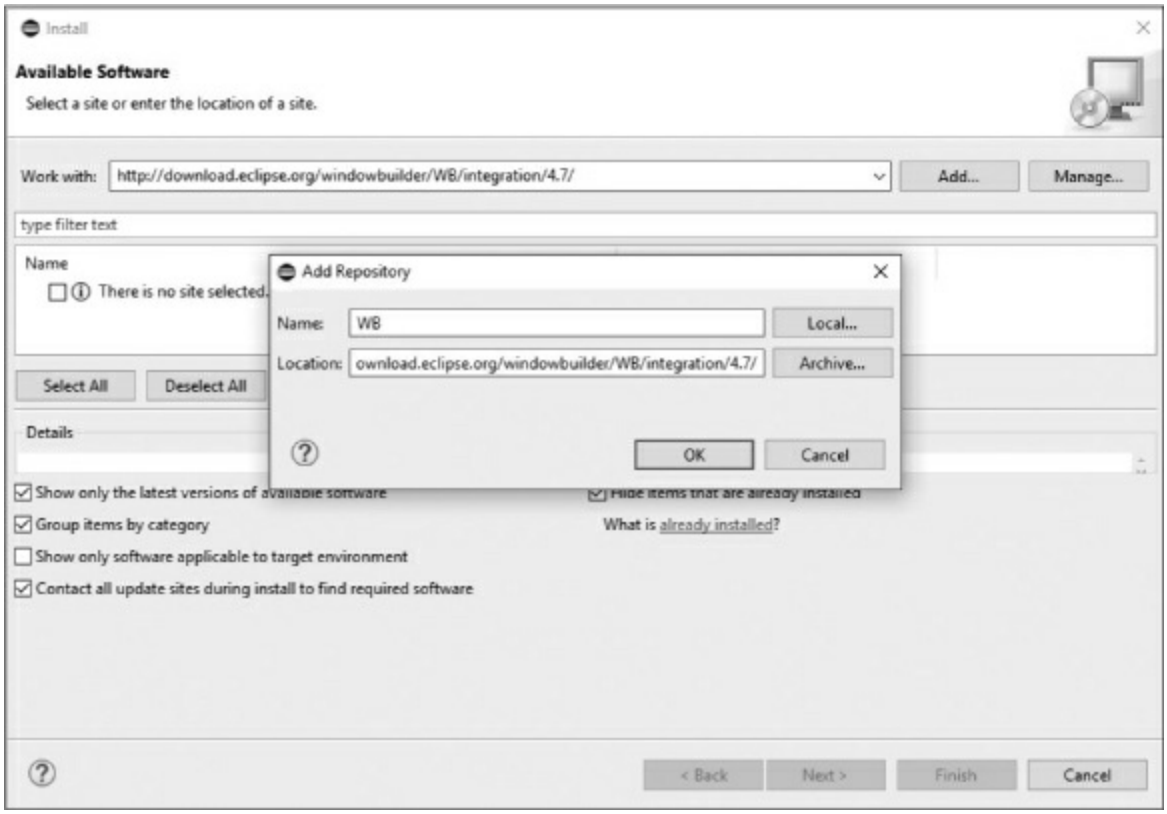


Figure 1-6: Use Eclipse's Install New Software window to add the WindowBuilder Editor.

The software may take a few minutes to install—you'll see a progress indicator in the bottom-right corner of Eclipse. When installation completes, you'll be asked to restart Eclipse. Click **Restart Now**, and the WindowBuilder installation will be complete.

Next, let's add a few optional tweaks to make code in Eclipse easier to read by changing the background, text colors, and font.

Customizing Eclipse's Look and Feel

Now that you've installed the required elements, you may also want to customize Eclipse's appearance. On Windows and Linux, you can access the Preferences menu by going to **Window ▸ Preferences**. On macOS, go to the menu bar and select **Eclipse ▸ Preferences**.

For example, you may want to change the *theme* (color palette) and the *font size* used in the text editor. Depending on your monitor's size and the environment in which you're programming, your theme and your font size can make a big difference in readability, comfort, and even productivity.

In the Preferences menu, under **General ▸ Appearance**, you'll see the

Theme option. You can choose from favorites such as Classic (light, grayish background with dark fonts) or Dark (black background with brighter, more colorful fonts). I prefer the Dark theme because the font colors are brighter against the dark background and the display can be easier to read on a monitor or projection screen.

Under **General ▸ Appearance ▸ Colors and Fonts**, you can change the font size. In the Colors and Fonts pane on the right, choose **Basic ▸ Text Font** and click the **Edit** button. The Font menu will pop up. Choose a font you can read easily. I prefer Courier New or Consolas. I recommend a font size of 18 to 20 and a font style of Bold. The Font window will display a sample of the selected font, style, and size, as shown in Figure 1-7.

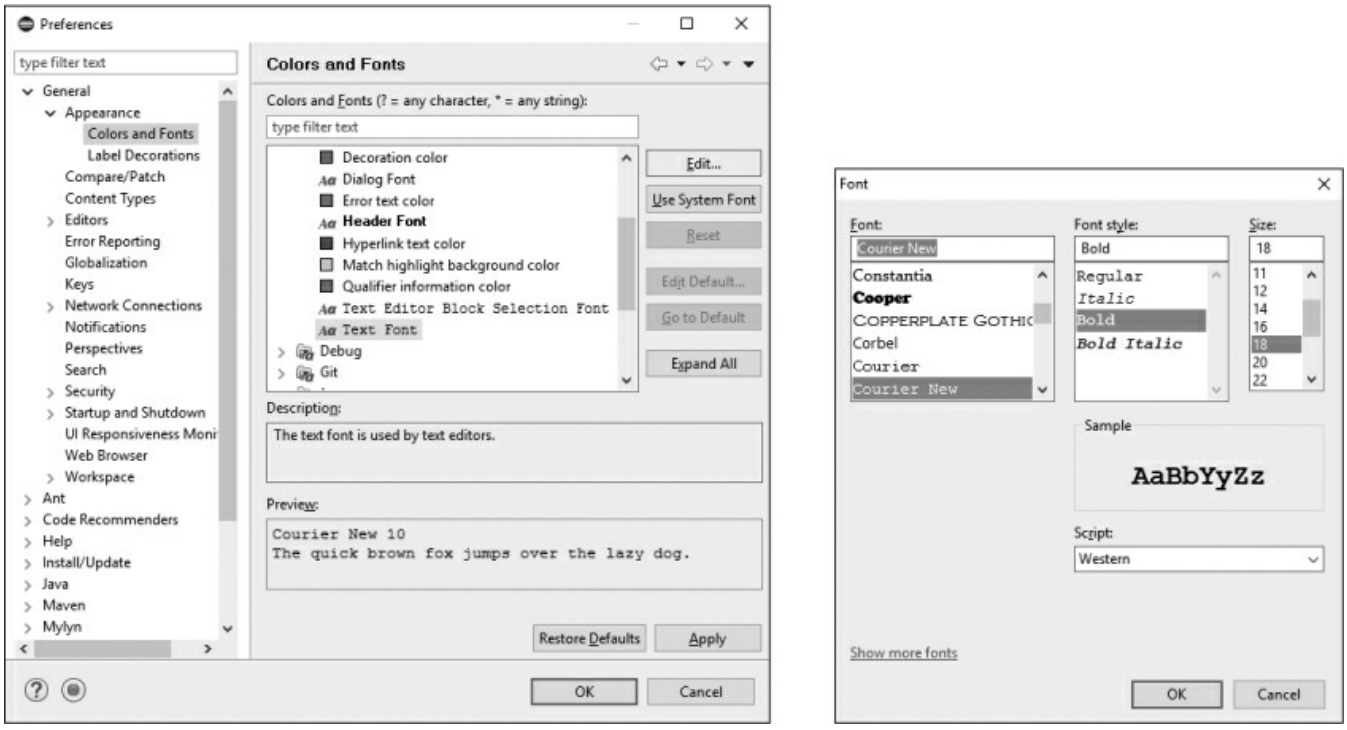


Figure 1-7: The Colors and Fonts preferences menu (left) and the Font preferences menu (right)

Click **OK** when you're finished with your selections, and you'll return to the main Eclipse workspace. You'll see your preference settings in effect when you begin coding in the Java text editor in Chapter 2.

Installing Android Studio for Mobile App Development

Android Studio is the official Android mobile app development environment. It gives us the ability to design and code mobile apps using Android's native Java programming language. Like Java and Eclipse, Android Studio is free to download, install, and use. Because of its large size, Android Studio may take from several minutes to several hours to download and install, depending on your internet connection.

To download Android Studio, go to *http://developer.android.com/studio/* and click **Download Android Studio**, as shown in Figure 1-8. Read and agree to the terms and conditions and then click **Download Android Studio for <operating system>**.

Follow the instructions to complete the installation. The installation process may download additional *software development kit (SDK)* components, and it could take several more minutes to install once all the components are downloaded.

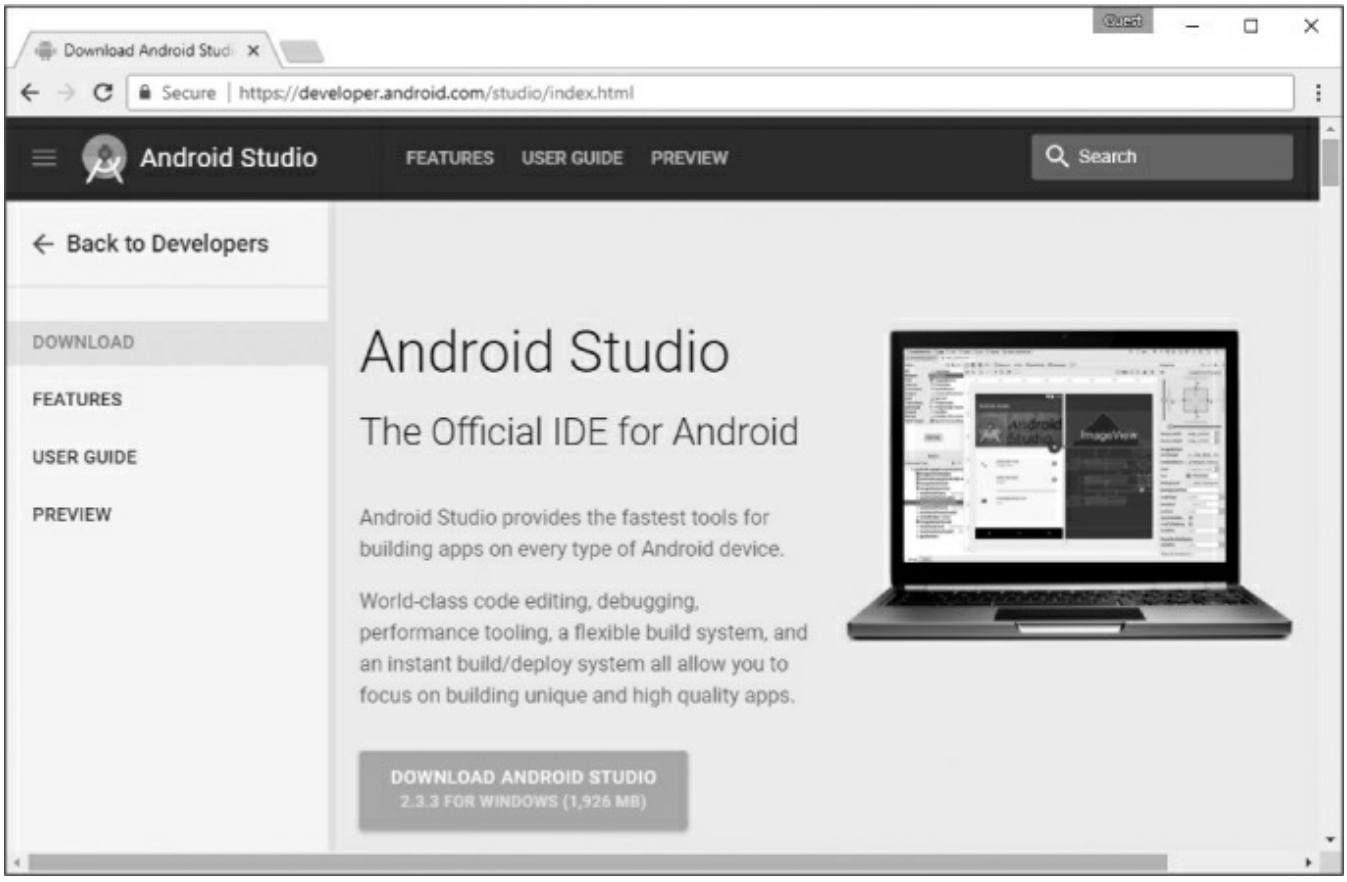

Download Android Studi
Secure | https://developer.android.com/studio/index.html
Android Studio
FEATURES
USER GUIDE
PREVIEW
Search
Back to Developers
DOWNLOAD
FEATURES
USER GUIDE
PREVIEW
Android Studio
The Official IDE for Android
Android Studio provides the fastest tools for building apps on every type of Android device.
World-class code editing, debugging, performance tooling, a flexible build system, and an instant build/deploy system all allow you to focus on building unique and high quality apps.
DOWNLOAD ANDROID STUDIO
2.3.3 FOR WINDOWS (1,926 MB)

Figure 1-8: Android Studio is the official Android mobile app development environment.

Getting to Know Java with JShell

Now that we've set up our programming environment, let's test our setup by using JShell, an *interactive interpreter* for Java. JShell is a great way to start learning how Java works, as it gives you feedback on your code right away. Before Java 9, Java coders had to type a complete program, compile it, and run it to see the result. Now, with JShell, we can type a single line of Java code, such as **`System.out.println("Hello, Java!")`**, press ENTER, and see the output immediately on the screen, as shown in Figure 1-9.

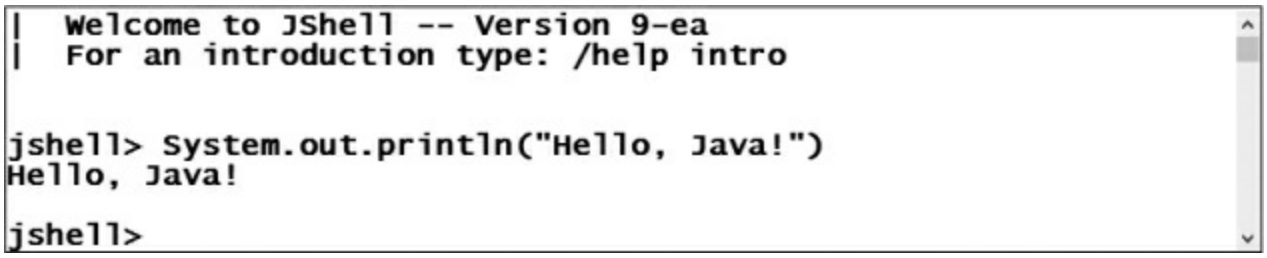

| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro
jshell> System.out.println("Hello, Java!")
Hello, Java!
jshell>

Figure 1-9: JShell in Java 9 allows us to quickly test code in an interactive command line shell.

Any valid Java statement will work in JShell, which makes it a great tool for learning the basics of Java coding, so let's jump right in!

Running JShell

You should be able to run JShell either from the command line or by setting up a shortcut. Just in case one of those options doesn't work for you, though, we'll go over both.

First, you must have JDK 9 installed. You can confirm that JDK 9 is installed by running a command from the command line.

Here's how to launch the command line interface on Windows, macOS, and Linux:

- On Windows, open the command prompt by going to the Start menu and typing **`cmd`** in the Search box. Press ENTER or click the Command Prompt icon.
- On macOS, open the Launchpad and enter **`terminal`** in the search box. Click the Terminal app icon.
- On Linux, search for **`terminal`** and click the Terminal app icon.

A command prompt window will appear. Enter **`java -version`** at the prompt, and Java will tell you which JDK version is installed. You'll see something like the following output on Windows:

```
C:\Users\Payne> java -version
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+153)
Java HotSpot(TM) 64-Bit Server VM (build 9-ea+153, mixed mode)
```

Or you'll see this output on macOS and Linux:

```
Payne:~ payne$ java -version
java version "1.9.0_33"
Java(TM) SE Runtime Environment (build 1.9.0_33)...
```

If Java responds with something containing version 9 or version 1.9, as shown here, you're ready to run JShell. If you'd like to run it from the command line, continue to the next section, "Running JShell from the Command Line." If you get an earlier version of Java, like 1.8, go back to "Installing Java 8 and 9 for Developers" on page 2 and install JDK 9. If version 9 doesn't appear on the command line after installing or if you'd prefer to set up a desktop shortcut, skip to "Running JShell from a Shortcut" on page 11.

Running JShell from the Command Line

To run JShell from the command prompt, type **`jshell`** and press ENTER. Java will respond with a JShell welcome message and prompt, which look like the following in Windows.

```
C:\Users\Payne> jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro

jshell>
```

On macOS or Linux, the message and prompt look similar:

```
Payne:~ payne$ jshell
|  Welcome to JShell -- Version 9-ea
|  For an introduction type: /help intro

jshell>
```

If you get the `jshell>` prompt, you're ready to move ahead to "Working with Java Expressions in JShell" on page 12. If the `jshell` command doesn't work on your computer after installing JDK 9, try following the steps in the next section.

Running JShell from a Shortcut

If you have any trouble getting JShell to run from the command line, or if you'd prefer to run it from a desktop shortcut, follow these instructions to access JShell directly from the JDK 9 *bin* folder and set up a shortcut. (The folder name *bin* is short for *binaries*, which are programs written in computer language with just 1s and 0s.)

> **NOTE**
>
> *Unfortunately, this method doesn't reliably work for Linux, but you should be able to use the command line method.*

You can find JShell in the following directories:

- On Windows, use *C:\Program Files\Java\jdk-9\bin\jshell.exe*
- On macOS, use */Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/bin/jshell*

Your JDK folder may be labeled *jdk-1.9.x* instead of *jdk-9*, but it will work the same either way. Navigate to the JDK 9 *bin* folder on your computer and find the JShell file, as shown in Figure 1-10.

You can double-click the JShell file's icon to run JShell. You can also create a shortcut to run JShell directly from your desktop for convenience:

- On Windows, right-click *jshell.exe* and select **Create Shortcut**. Windows will respond with "Windows can't create a shortcut here. Do you want the shortcut to be placed on the desktop instead?" Click **Yes**.
- On macOS, CONTROL-click the *jshell* file and select **Make Alias**. A file called *jshell alias* will appear. Drag it to your desktop.

Now, anytime you want to run JShell, simply double-click the icon on your desktop, and you're ready to code in Java.

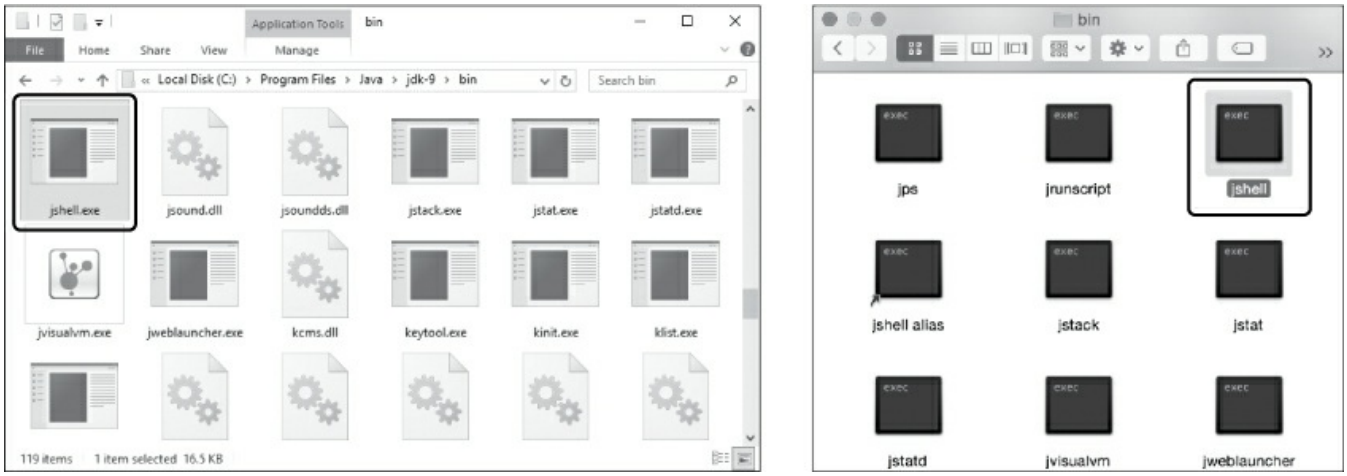


Figure 1-10: The JShell program file in the JDK 9 bin *folder on Windows (left) and macOS (right)*

Working with Java Expressions in JShell

An *expression* is any combination of values (like numbers or text) and operators that results in another value. An *operator* performs an operation like addition, subtraction, multiplication, or division. In Java, the symbols we use for these operators are, respectively, `+`, `-`, `*`, and `/`. Let's try one out with a simple math expression. At the JShell prompt, type **`2+2`** and press ENTER:

```
jshell> 2+2
$1 ==> 4
```

JShell tells you the answer: the value of the expression `2+2` is `4`. The `$1` is a *temporary variable*. JShell creates these variables to store values, and here the variable `$1` is storing the value `4` temporarily, in case you want to use it later. If you want to know what value is in a variable, you can enter it in JShell. For example, if you enter **`$1`** now, JShell will tell you `$1` is storing the value `4`:

```
jshell> $1
$1 ==> 4
```

Let's try another expression. This time, let's combine two strings of text. *Strings* are characters between quotation marks that are used to display words, names, and other text. You can use the `+` operator to combine strings:

```
jshell> "Your" + "Name"
$3 ==> "YourName"
```

You'll see that a different temporary variable is created—in this case, `$3`. The number after the `$` indicates the line of code where the expression was entered.

This is my third snippet of code, so JShell is storing `"YourName"` in the variable `$3`.

Notice also that Java *concatenates*, or adds, the two strings, with no space between. If we want a space between two concatenated strings, we have to include it inside double quotes, like this:

```
jshell> "Your" + " " + "Name"
$4 ==> "Your Name"
```

Any valid Java expression can be evaluated in JShell, so go ahead and try a few more expressions. To edit a statement you've entered, press the up arrow key. JShell will display the last command you typed, allowing you to edit it and press ENTER to run the statement again. Pressing the up arrow more than once will scroll you through the history of previous statements, all the way to the first line you entered.

Declaring Java Variables in JShell

Evaluating expressions with simple values is nice, but you'll usually want to store values in variables so you can use them later. JShell has been creating temporary variables for us automatically, like `$1` and `$3` in the previous section, but you can also create, or *declare*, your own.

Numeric Variables

Let's create an integer (whole number) variable called `x` and store the value `42` in it. Java stores whole numbers in variables of the type `int`, so at the JShell prompt, enter **`int x = 42`**:

```
jshell> int x = 42
x ==> 42
```

The equal sign (`=`) is the *assignment operator* in Java, meaning that it is used to *assign* a value to a variable. JShell responds, letting us know the variable `x` contains the value `42`. Now, when we want to use that value, we can just call the variable by name. Let's find out what **`x * 2`** (`x` times two) equals:

```
jshell> x * 2
$6 ==> 84
```

Java uses the asterisk (`*`) as the symbol for multiplication, and JShell is letting

us know that `x * 2` is equal to `84`. Did we change the value of `x`, though, by getting its value and multiplying that value by two? Let's find out by entering **`x`**:

```
jshell> x
x ==> 42
```

Whew! Our variable `x` still has the same value, `42`. That means we can use the value stored in a variable without changing the variable.

So how would we change the value stored in the variable? We simply use the assignment operator again. Enter **`x = x + 7`** at the JShell prompt:

```
jshell> x = x + 7
x ==> 49
```

We've *replaced* the value stored in `x` by taking the old `x` value and adding `7` to it. From now on, whenever we ask for the value of `x`, we'll get `49`, until we change it again. The value of a variable can change, or *vary*, as often as we need it to—that's why they're called variables.

Let's try a few variables of different types. We've seen an integer value, so let's try a decimal, or *floating-point*, number. Java stores decimal numbers in variables of the type `double` (short for *double-precision floating point*), so create a `double` called `meters` and store the decimal value `1.83` in it:

```
jshell> double meters = 1.83
meters ==> 1.83
```

Java handles decimal values just as easily as integer values. Let's do some math and convert meters to centimeters:

```
jshell> double centimeters = meters * 100
centimeters ==> 183.0
```

We've converted the value in `meters` to a value in `centimeters` by multiplying it by `100`.

Java handles a few other types of numbers, but `int` and `double` values are the most common ones you'll use. Anytime you come across a new type of variable, feel free to open JShell and play around with different values.

String Variables

The `String` variable type is used to store strings of text characters. Let's define a `String` variable called `myName` and store a name in it as follows (you can use your name instead of mine):

```
jshell> String myName = "Bryson Payne"
myName ==> "Bryson Payne"
```

We use the equal sign assignment operator, like we did with numeric variables.

> **NOTE**
>
> *Case matters in Java variable, method, and class names. All-lowercase* `myname` *is different from all-uppercase* `MYNAME`*, and both of those are different from* `myName`*. The convention, or common way of doing things, in Java is to use* camel case *—capitalizing the first letter of each new word in a name, like* `myName` *or* `thisIsASillyNameButShowsCamelCase`*, so that the words resemble a camel's humps. Classes are in camel case and also begin with an uppercase letter.*

Now let's use the value stored in `myName`. Let's say you've added a certification or earned a degree. Add some characters after your name, as follows:

```
jshell> myName + ", PhD"
$12 ==> "Bryson Payne, PhD"
```

Notice that we didn't use the assignment operator, so the value stored in `myName` should still be your name without the extra characters.

This time, let's change the value stored in the variable `myName`. Add a formal title or greeting to your name, like you would for addressing an envelope or invitation:

```
jshell> myName = "Dr. " + myName
myName ==> "Dr. Bryson Payne"
```

JShell shows us the updated value stored in `myName`. We'll continue to work with both numeric and text variables in the next section, and you'll learn to output values to the screen from inside a Java program.

Printing Output in Java

Up to this point, we've just been evaluating expressions by entering them into JShell to see their values, but this won't usually be the case when we start writing actual programs. In a Java program, we won't see anything echoed back to the screen as we type line by line.

When we want to print something to the screen, we use a print function, like `System.out.println()`, which prints a line of output to the system console, or your screen. If you're still using the same JShell window from the previous section, you can print the value we stored in `x` as follows:

```
jshell> System.out.println(x)
49
```

If you get an error, declare a new variable `int x = 49` and then run the `print` statement again.

Notice that this time, JShell doesn't respond with `x ==> 49`, because you aren't asking it to evaluate an expression. The `println()` statement tells JShell to print only what you've placed inside the parentheses, which is the variable `x`, so JShell simply responds with `49`.

Let's try some output using strings instead. Enter the following statement:

```
jshell> System.out.println("Hello, " + myName)
Hello, Dr. Bryson Payne
```

As long as you've still got a variable called `myName` from the previous section, Java will greet you by name.

Whenever you need to print information to the screen for a user to read, the `System.out.println()` statement will allow you to print exactly what you want to communicate.

JShell Commands

JShell has been so easy to use that you might never want to leave it, but eventually you'll need to get on with other work, like building exciting desktop and mobile apps in Java. Let's take a look at the commands that JShell has to offer, including the command to exit JShell.

At the JShell prompt, enter **`/help`**:

```
jshell> /help
```

JShell will respond with the full list of special commands that it recognizes. We precede each command with the forward slash, /, to indicate that we're talking to the JShell program directly, not to Java. Here's a shortened, partial version of the commands JShell recognizes:

```
|    /list [<name or id>|-all|-start]     -- list the source you have typed
|    /edit <name or id>                   -- edit a source entry referenced by
name or id
|    /save [-all|-history|-start] <file>  -- save snippet source to a file
|    /open <file>                         -- open a file as source input
|    /vars [<name or id>|-all|-start]     -- list the declared variables and
their values
|    /imports                             -- list the imported items
|    /exit                                -- exit jshell
|    /reset                               -- reset jshell
|    /history                             -- history of what you have typed
|    /help [<command>|<subject>]          -- get information about jshell
```

Try some of the commands, like `/list` to see all the source code you've typed. You'll notice that JShell adds semicolons at the ends of lines where needed—the semicolon is how Java separates multiple statements in a program. The `/history` command shows you what you've typed, including commands like `/help`, `/list`, and even `/history`.

When you're editing regular Java programs in the next chapter and beyond, you'll be working in files, saving often, and reopening the files when you want to work on them more. In JShell, though, once you close the JShell window, everything you've typed is gone for good—unless you save it, that is. Thankfully, JShell includes the ability to save, open, and edit the awesome code snippets you produce. To save code you created in JShell, use the `/save` command and specify where you'd like to store your file:

```
jshell> /save ~/Desktop/filename.txt
```

The tilde (~) means your user directory on your computer, so this command will save all the code you've typed since you opened JShell, or since the last `/reset`, in a file on your desktop. Look at your computer's desktop screen and you should see the new file.

To open the file again, use the `/open` command and tell JShell where to find it:

```
jshell > /open ~/Desktop/filename .txt
```

JShell will open the file and run the code.

Anytime you write a snippet of code that you'd like to store for later use, use the `/save` and `/open` commands to keep your code from one JShell session to the next.

To start a new code snippet, use the `/reset` command. JShell will remember and save only what you type after the `/reset` command, but you can reopen a file that you've saved anytime. Try the following code for a quick example of saving a file, resetting, and opening a file:

```
❶ jshell> /reset
  |  Resetting state.
❷ jshell> System.out.println("Hello, Java!")
  Hello, Java!
  jshell> System.out.println("My name is Bryson, nice to meet you!")
  My name is Bryson, nice to meet you!
❸ jshell> /save ~/Desktop/myJava.txt
❹ jshell> /reset
  |  Resetting state.
❺ jshell > /open ~/Desktop/myJava.txt
❻ Hello, Java!
  My name is Bryson, nice to meet you!
```

NOTE

The numbered circles point out important lines, but they aren't actually part of the code.

First, you need to reset JShell ❶ so that you don't include any earlier code you might have written before the code you want to save. Once you've done that, you can write out your program, which in this case is two `print` statements ❷, and then save it to your desktop ❸. Even after JShell is reset ❹, the `/open` command ❺ recalls the two lines of code we typed before the `/save` command and runs them ❻. You can use `/edit` to change the code and `/save` again when you're done.

JShell can store and recall a snippet whether it's a single line or as many lines as you care to type.

If you're finished exploring, use the `/exit` command to close JShell:

```
jshell> /exit
```

JShell is even polite when you leave, bidding you a kind goodbye.

What You Learned

In this chapter, you've not only installed Java, Eclipse, and AndroidStudio but you've also begun learning Java by testing various commands in the JShell interactive shell. JShell is an exciting improvement to the already omnipresent, industry-standard Java, and it's a great tool for beginners and seasoned developers alike.

For new coders, JShell encourages exploration and removes some of the unnecessary barriers to learning basic Java commands. For experienced programmers, JShell provides a way to rapidly test new pieces of code and see results instantly on the screen. As an educator and coder, I'm excited about what JShell means for the future of Java and for the millions of coders who'll be able to benefit from this important new set of tools.

You've set up your programming environments for Java desktop and mobile apps, and you've learned how to test code in JShell. You've also gained some experience with a number of programming concepts that you'll use in the Java desktop and mobile apps that we'll build in the upcoming chapters. Here's a quick overview of what you've accomplished so far:

- Installing Java JDK 8 and 9
- Installing Eclipse IDE for Java Developers and the WindowBuilder Editor
- Installing Android Studio for mobile app development
- Running JShell from the command line and from the *JDK9/bin* folder
- Evaluating Java expressions in JShell, including numbers and text
- Declaring integer, decimal, and string variables in Java
- Printing output to the screen using `System.out.println()`
- Using JShell commands such as `/reset`, `/edit`, `/save`, `/open`, and `/exit`
- Saving and opening files from JShell

In the next chapter, we'll build our first complete Java application, the Hi-Lo

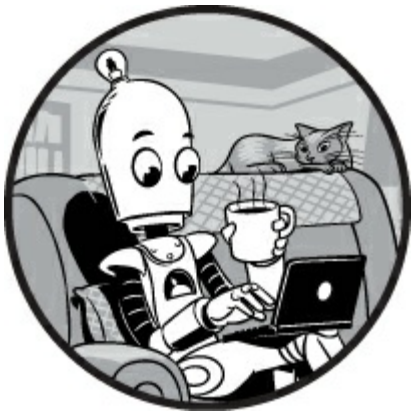
guessing game. In the chapters that follow, we'll create text-based programs, desktop apps, and, finally, Android mobile apps for each of our programs.

Along the way, we'll use the programming concepts we picked up in JShell, including expressions, variables, output, and more. Whether you're building your first app or your thousandth, a desktop app for work or a mobile game for fun, these same elements will be part of everything you code.

You've got the tools, so let's jump into Chapter 2 and get started!

2

BUILD A HI-LO GUESSING GAME APP!



Let's begin by coding a fun, playable game in Java: the Hi-Lo guessing game. We'll program this game as a *command line application*, which is just a fancy way of saying it's text based (see Figure 2-1). When the program runs, the prompt will ask the user to guess a number between 1 and 100. Each time they guess, the program will tell them whether the guess is too high, too low, or correct.

```
Command Prompt - java HiLo
Guess a number between 1 and 100:
50
50 is too high. Try again.
Guess a number between 1 and 100:
25
25 is too low. Try again.
Guess a number between 1 and 100:
37
37 is too low. Try again.
Guess a number between 1 and 100:
43
43 is correct! You win!
It only took you 4 tries! Good work!
Would you like to play again (y/n)?
y
```

Figure 2-1: A text-based Hi-Lo guessing game

Now that you know how the game works, all you have to do is code the steps to play it. We'll start by mapping out the app at a high level and then code a very simple version of the game. By starting out with a goal in mind and

understanding how to play the game, you'll be able to pick up coding skills more easily, and you'll learn them with a purpose. You can also enjoy the game immediately after you finish coding it.

Planning the Game Step-by-Step

Let's think about all the steps we'll need to code in order to get the Hi-Lo guessing game to work. A basic version of the game will need to do the following:

1. Generate a random number between 1 and 100 for the user to guess.
2. Display a *prompt*, or a line of text, asking the user to guess a number in that range.
3. Accept the user's guess as input.
4. Compare the user's guess to the computer's number to see if the guess is too high, too low, or correct.
5. Display the results on the screen.
6. Prompt the user to guess another number until they guess correctly.
7. Ask the user if they'd like to play again.

We'll start with this basic structure. In Programming Challenge #2 on page 40, you'll try adding an extra feature, to tell the user how many tries it took to guess the number correctly.

Creating a New Java Project

The first step in coding a new Java app in Eclipse is creating a project. On the menu bar in Eclipse, go to **File ▸ New ▸ Java Project** (or select **File ▸ New ▸ Project**, then **Java ▸ Java Project** in the New Project wizard). The New Java Project dialog should pop up, as shown in Figure 2-2.

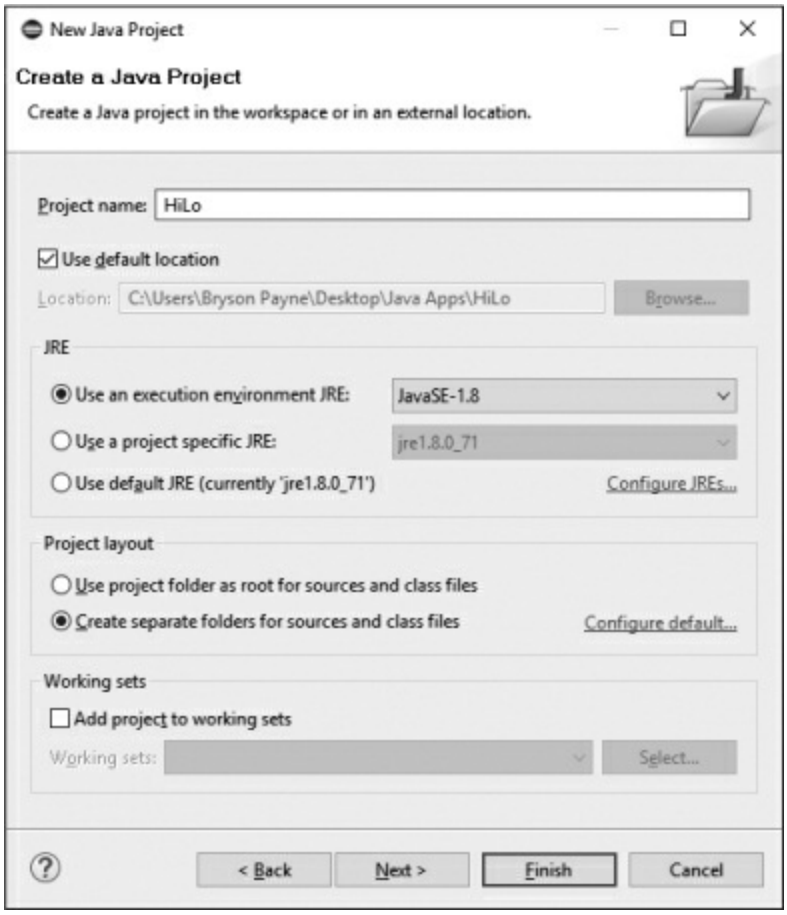


Figure 2-2: The New Java Project dialog for the Hi-Lo guessing game app

Type `HiLo` into the Project name field. Note that uppercase and lowercase letters are important in Java, and we'll get in the habit of using uppercase letters to start all of our project, file, and class names, which is a common Java practice. And we'll use camel case, as *Hi* and *Lo* are two words: *HiLo*. Leave all the other settings unchanged and click **Finish**. Depending on your version of Eclipse, you may be asked if you want to open the project using the Java Perspective. A *perspective* in Eclipse is a workspace set up for coding in a specific language. Click **Yes** to tell Eclipse you'd like the workspace set up for convenient coding in Java.

Creating the HiLo Class

Java is an *object-oriented programming language*. Object-oriented programming languages use *classes* to design reusable pieces of programming code. Classes are like templates that make it easier to create *objects*, or instances of that class. If you think of a class as a cookie cutter, objects are the cookies. And, just like a cookie cutter, classes are reusable, so once we've built a useful class, we can reuse it over

and over to create as many objects as we want.

The Hi-Lo guessing game will have a single class file that creates a guessing game object with all the code needed to play the game. We'll call our new class `HiLo`. The capitalization matters, and naming the class `HiLo` follows several Java naming conventions. It's common practice to start all class names with an uppercase letter, so we use a capital `H` in `HiLo`. Also, there should be no spaces, hyphens, or special characters between words in a class name. Finally, we use camel case for class names with multiple words, beginning each new word with a capital letter, as in `HiLo`, `GuessingGame`, and `BubbleDrawApp`.

To create the new `HiLo` class, first find your *HiLo* project folder under the Package Explorer pane on the left side of the Eclipse workspace. Expand the folder by clicking the small arrow to the left of it. You should see a subfolder called *src*, short for *source code*. All the text files containing your Java programs will go in this *src* folder.

Right-click the *src* folder and select **New ▸ Class**, as shown in Figure 2-3.

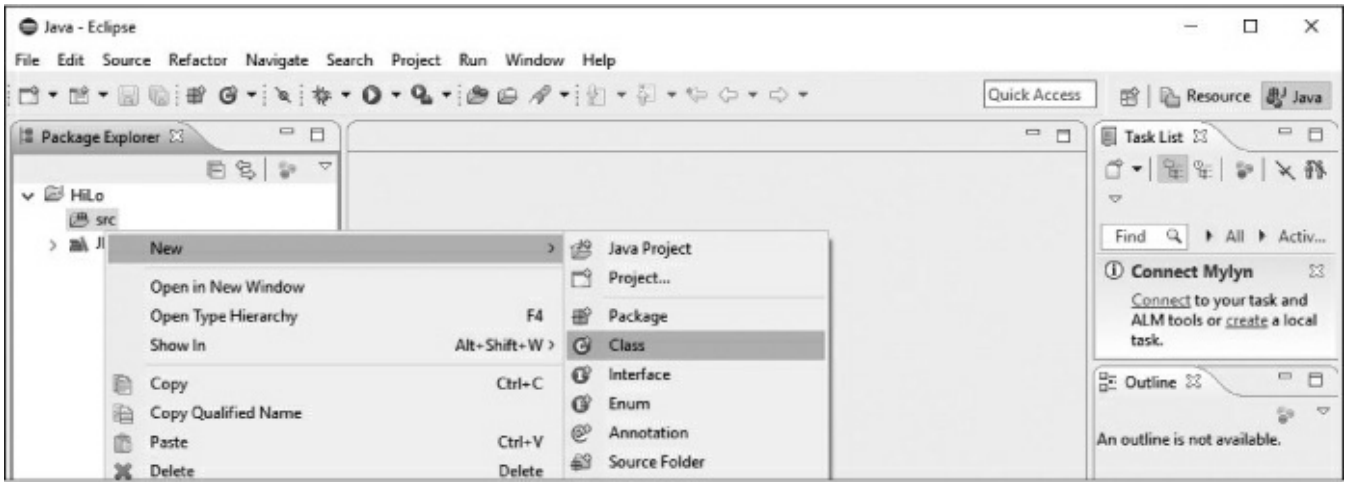


Figure 2-3: Creating a new class file for the Hi-Lo guessing game app

The New Java Class dialog will appear, as shown in Figure 2-4. Type **`HiLo`** into the Name field. Then, under *Which method stubs would you like to create?*, check the box for **public static void main(String[] args)**. This tells Eclipse that we're planning to write a `main()` program method, so Eclipse will include a *stub*, or skeleton, for the `main()` method that we can fill in with our own code. *Methods* are the functions in an object or class. The `main()` method is required any time you want to run an app as a stand-alone program.

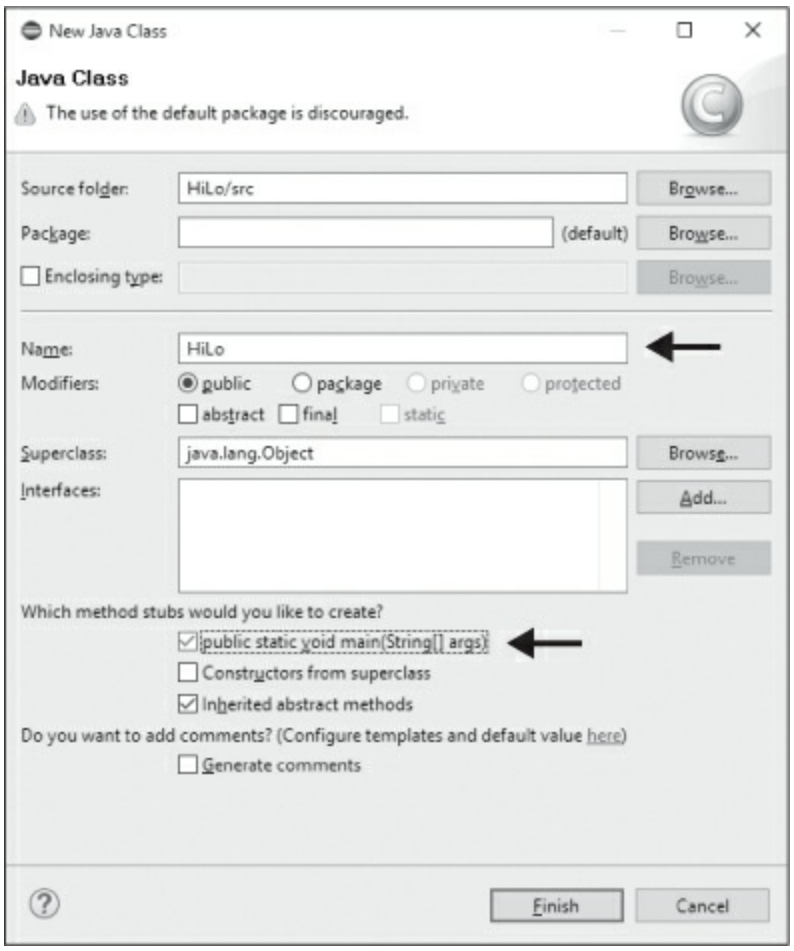


Figure 2-4: Name the new Java class HiLo *and select the checkbox to create a main() method.*

Click **Finish** in the New Java Class dialog, and you should see a new file named *HiLo.java* that contains the code shown in Listing 2-1. This Java file will be the outline of the Hi-Lo guessing game. We'll write the guessing game program by editing this file and adding code inside it.

```
❶ public class HiLo {
    ❷ public static void main(String[] args) {
          // TODO Auto-generated method stub

      }
}
```

Listing 2-1: The stub code for the HiLo guessing game class, generated by Eclipse

Eclipse creates this code all on its own. The class `HiLo` is `public` ❶, meaning we can run it from the command line or terminal.

Java groups statements with braces, `{` and `}`. The opening brace, `{`, begins a block of statements that will form the body of the `HiLo` class. The closing brace,

}, ends the block of statements. Inside the class is the `main()` method ❷, which is the method that will run when the class is executed.

Inside the opening brace for the `main()` method is a comment line that starts with two forward slashes, `//`. Comments are for us (the humans) to read. They're ignored by the computer, so we can use them to help us remember what a section of code does or to leave notes for future use. You can delete the `TODO` comment in Listing 2-1.

Generating a Random Number

The first programming task for our game is to generate a random number. We'll use the `Math` class, which contains a method for generating a random floating-point (decimal) number between 0.0 and 1.0. Then, we'll convert that decimal value to an *integer* (a whole number) between 1 and 100. The `Math` class is a built-in class that contains many useful math functions like the ones you might find on a nice scientific calculator.

Inside the `main()` method, add the comment and line of code shown in Listing 2-2. (The new code is shown in black and the existing code is shown in gray.)

```
public class HiLo {
    public static void main(String[] args) {
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
    }
}
```

Listing 2-2: The code to create a random number between 1 and 100

First, we need to create a variable to hold the random number the user will be trying to guess in the app. Since the app will ask the user to guess a whole number between 1 and 100, we'll use the `int` type, short for *integer*. We name our variable `theNumber`. The equal sign, `=`, assigns a value to our new `theNumber` variable. We use the built-in `Math.random()` function to generate a random number between 0.0 and just under 1.0 (0.99999). Because `Math.random()` generates numbers only in that specific range, we need to multiply the random number we get by 100 to stretch the range from 0.0 to just under 100.0 (99.99999 or so). Then we add 1 to that value to ensure the number runs from 1.0 (0.0 + 1) to 100.99999.

The `(int)` part is called a *type cast*, or just *cast* for short. Casting changes the

type of the number from a decimal number to an integer. In this case, everything after the decimal point is removed, resulting in a whole number between 1 and 100. Java then stores that number in the variable `theNumber`, the number the user is trying to guess in the game. Finally, we add a semicolon (;) to indicate the end of the instruction.

Now, you can add a `System.out.println()` statement to print the number you've generated:

```
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
    }
}
```

After we add this line of code, we can run the program to see it generate and print a random number. Click the green run button in the top menu bar to compile and run the program, as shown in Figure 2-5. You can also go to the **Run** menu and select **Run**.

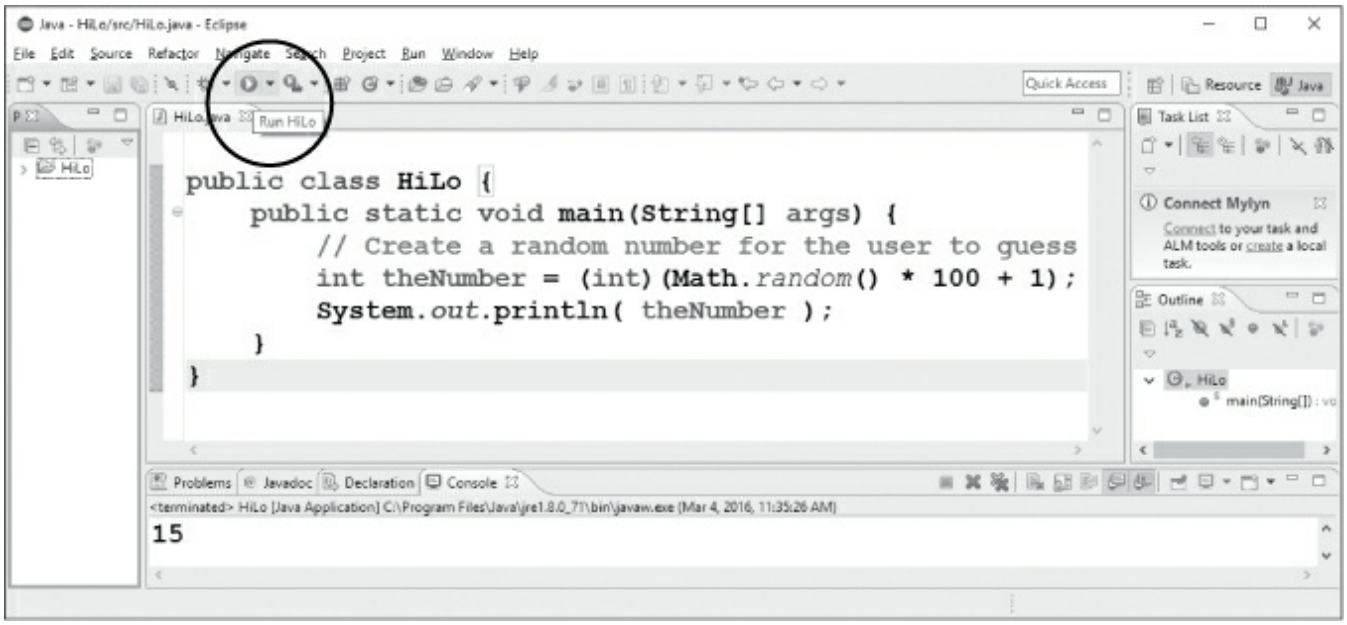

Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Run HiLo
public class HiLo {
public static void main(String[] args) {
// Create a random number for the user to guess
int theNumber = (int)(Math.random() * 100 + 1);
System.out.println(theNumber);
}
}
Problems Javadoc Declaration Console
<terminated> HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 4, 2016, 11:35:26 AM)
15

Figure 2-5: Printing a random number to the screen

Your random number will appear in the small console window at the bottom of the screen, as shown in Figure 2-5. If you run your program again, you'll see a different number between 1 and 100.

This would be a great time to play with the program a bit. Try generating a number between 1 and 10, or 1 and 1,000—even 1 to 1,000,000. Java will accept numbers all the way to a billion or so. Just remember to write your numbers

without commas: 1,000 becomes `1000` in Java, and 1,000,000 is written `1000000`. You probably don't want to guess a number between 1 and 1,000,000 the first time you play the game, though, so remember to change this line back before you move ahead.

NOTE

Remember to save your code often. Eclipse will save for you automatically every time you run a program, but it's a good idea to save after every few lines of code. In fact, pressing CTRL-S (or ⌘-S) to save after each line of code isn't a bad habit to get into. I've never heard a coder say they wish they hadn't saved so often, but I've experienced losing unsaved code a few times myself, and it's not fun. Save often, and remember that you can use ***Edit ▸ Undo*** *if you ever type something incorrectly or accidentally delete a section of code.*

Getting User Input from the Keyboard

Now let's add the code that allows the user to guess a number. To do this, we'll need to *import* some additional Java capabilities. Java comes with many libraries and packages that we can use in our own projects. Libraries and packages are sets of code that someone else has created. When we import them, we get new features that make creating our own programs even easier. We can access packages and libraries whenever we need them using the `import` statement.

For the guessing game program, we need to be able to accept keyboard input from the user. The `Scanner` class, contained in the `java.util` utilities package, provides several useful functions for working with keyboard input. Let's import the `Scanner` class into our program. Add the following statement at the top of the *HiLo.java* file, before the line `public class HiLo`:

```
import java.util.Scanner;

public class HiLo {
```

This line imports the `Scanner` class and all its functionality from the main Java utilities package. The `Scanner` class includes functions like `nextLine()` to accept a line of input from the keyboard and `nextInt()` to turn text input from the keyboard into an integer number that can be compared or used in calculations. To use the `Scanner` class for keyboard input, we have to tell it to use the keyboard

as its source.

We want to do this before anything else in the program, so add this line of code inside the top of the `main()` method:

```
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
```

This line creates a `Scanner` object called `scan` that pulls input from the computer's keyboard, `System.in`.

Although this new line of code sets up the `scan` object, it doesn't actually ask for input yet. To get the user to type in a guess, we'll need to prompt them by asking them to enter a number. Then, we'll take the number they enter from the keyboard and store it in a variable that we can compare against `theNumber`, the computer's original random number. Let's call the variable that will store the user's guess something easy to remember, like `guess`. Add the following line next:

```
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
        int guess = 0;
```

This statement both *declares* a variable called `guess` of type `int` (an integer in Java), and it *initializes* the `guess` variable to a starting value of `0`. Some programming languages require a variable to be declared and then initialized in separate lines of code, but Java allows programmers to include both the declaration and initialization of variables in a single line. Java requires every variable to be declared with a specific *type*, or kind of information it should store. The user's guess will be a whole number, so we've declared `guess` as an `int`.

Next, we need to prompt the user to enter a guess. We can let the user know the program is ready for input by printing a line of text to the console window (or command line). We access this text-based screen as a part of our computer system through the `System` class, just like we did for keyboard input. But this time, we want to *output* information for the user to read. The object that lets us access the command line console for output is `System.out`. Similar to the `System.in` object that allows us to receive text input from the keyboard, `System.out`

gives us the ability to output text to the screen. The specific function to print a line of text is the `println()` command:

```
    int guess = 0;
    System.out.println("Guess a number between 1 and 100:");
```

Here we are using *dot notation*, which lists a class or object, followed by a dot and then a method or an attribute of that class or object. Methods need to be called with dot notation to tell Java which object or class they belong to; for example, `Math.random()`. *Attributes* are the values stored in an object or class.

For example, `System` is a class representing your computer system. `System.out` is the command line screen object contained in the `System` class, because your computer monitor is part of your overall computer system. `System.out.println()` is a method to print a line of text using the `System.out` object. We'll get more practice using dot notation as we continue.

Now that the user knows what kind of input the program is expecting from them, it's time to check the keyboard for their guess. We'll use the `Scanner` object called `scan` that we created earlier. Scanners have a method called `nextInt()` that looks for the next `int` value the user inputs from the keyboard. We'll store the user's guess in the variable `guess` that we created earlier:

```
    System.out.println("Guess a number between 1 and 100:");
    guess = scan.nextInt();
```

This statement will wait for the user to type something into the console window (hopefully a whole number between 1 and 100—we'll see how to make sure the user enters a valid number in Chapter 3) and press ENTER. The `nextInt()` method will take the string of text characters the user entered (`"50"`, for example), turn it into the correct numeric value (`50`), and then store that number in the variable `guess`. Take a moment to save the changes you've made so far.

Making the Program Print Output

We can also check to make sure our program is working so far by adding another `println()` statement:

```
    guess = scan.nextInt();
    System.out.println("You entered " + guess + ".");
```

This line uses the `System.out.println()` method again, but now we're

combining text and numeric output. If the user guesses `50`, we want the output to read, `"You entered 50."` To make this happen, we form a `println()` statement that mixes text with the number stored in the variable `guess`.

Java allows us to concatenate strings of text using the `+` operator. We use double quotation marks to specify the text we want to output first (`"You entered "`). Note the space before the closing quotation marks—this tells th program that we want a space to appear in the printed output after the last word. Java ignores most spacing, but when a space is included inside the quotation marks of a string of text, it becomes part of that text.

We also want to print the number the user guessed. We've stored this value in the variable called `guess`, so we just have to use the `println()` statement to output that value. Fortunately, in Java, when you include a variable in a `println()` statement, Java prints the value contained in that variable. So, immediately after the text `"You entered "`, we add the concatenation operator (+) followed by the variable name `guess`. Finally, we want to end the sentence with a period, so we use another concatenation operator followed by the text we want, contained in double quotation marks, so it looks like `"."`.

Listing 2-3 puts together all of our lines of code so far.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
      ❶ // System.out.println( theNumber );
        int guess = 0;
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
  }
}
```

Listing 2-3: The code to this point generates a random number and allows the user to guess once.

At ❶, note that I turned `System.out.println( theNumber );` into a comment by adding a pair of forward slashes to the start of that line. This is called *commenting out*, and it's a useful technique for *debugging*—finding and fixing bugs or errors in

programs. We used this `println()` statement earlier to show the value of the variable `theNumber` while we were writing and testing the program. Now, rather than deleting the line entirely, we can turn it into a comment so it's ignored by the computer. If we want to use that line again, we can just remove the `//` to include it in the program.

Now let's save our program and run it to see how it works so far. To run it, press the green run button or go to **Run ▸ Run**. Right now, the user can guess only once and the program doesn't check whether they guessed correctly. So next, we'll add some code so that the user can guess more than one time, and then we'll learn how to test each guess against `theNumber`.

Loops: Ask, Check, Repeat

To give the user more than one chance to guess the number, we need to learn how to build a loop! In the guessing game program, we need to ask the user for a guess until they guess correctly. *Loops* give us the ability to repeat a set of steps over and over. In this section, we'll build a loop for the steps that prompt the user for a guess and accept the keyboard input.

Loops are very powerful programming tools, and they're one of the reasons computers are so valuable in our daily lives and in the business world—computers are really good at repeating the same task predictably. And, if they're programmed correctly, they can do this all day, every day, without making mistakes. You or I might get tired of telling someone their guess is too high or too low, but the computer never does. It will also never forget the number or tell the player their guess is too low or too high when it's actually not.

Let's tap into the power of loops with a `while` loop. A `while` loop repeats a set of statements as long as some *condition* is true. A condition is just something we can test. For example, in this program, we want to know whether the user correctly guessed the secret number. If they didn't guess correctly, we want to keep giving them a chance to guess again until they get it right.

To write a `while` loop, we need to know what condition we want to test for before repeating the loop each time. In the guessing game, we want the user to guess again as long as their `guess` isn't equal to the secret number `theNumber`. When the user's guess is equal to the secret number, the user wins and the game is over, so the loop should stop.

To create a `while` loop, we need to insert a `while` statement before the last three lines of code and then wrap the three lines for guessing inside a new pair of braces, as follows:

```
        int guess = 0;
        while (guess != theNumber) {
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
        }
    }
}
```

We use the keyword `while` to let Java know we're building a `while` loop, and then we put the appropriate condition inside parentheses. The part inside the parentheses, `guess != theNumber`, means that while the value stored in `guess` is not equal to (`!=`) the value stored in `theNumber`, the loop should repeat whatever statement or set of statements immediately follow this line of code. The `!=` operator is a *comparison operator*—in this case, it compares `guess` and `theNumber` and evaluates whether they're different, or *not equal*. You'll learn about other comparison operators in the next section, but this is the one we need for the guessing `while` loop.

We need to tell Java what statements to repeat in the `while` loop, so I've added an opening brace, `{`, after the `while` statement. In the same way that braces group all the statements together in the `main()` method, these braces group statements together inside the `while` loop.

There are three statements that we want to include inside the loop. First we need the `println()` statement that prompts the user to guess a number. Then we need the statement that scans the keyboard and records the guess with the `nextInt()` method. Finally, we need the `println()` statement that tells the user what they entered. To turn this set of statements into a block of code that will be run repeatedly in the `while` statement, we write the `while` statement and condition first, then an opening brace, then all three statements, and finally, a closing brace. Don't forget the closing brace! Your program won't run if it's missing.

One good programming practice that will help you keep your code organized and readable is using tab spacing correctly. Highlight the three statements inside the braces for the `while` statement and then press the TAB key to indent them.

The result should look like the following code:

```
        int guess = 0;
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            System.out.println("You entered " + guess + ".");
        }
    }
}
```

Correct indentation will help you remember to match up your opening and closing braces, and it will help you quickly see which statements are inside a loop or other block of code, as well as which statements are outside the loop. Indentation doesn't affect how your program runs, but if done well, it makes your program much easier to read and maintain.

Save your program now and run it to check that it works. The game is almost playable now, but we still need to tell the program to check if the user's guess is too high, too low, or just right. Time for (drum roll, please . . .) `if` statements!

if Statements: Testing for the Right Conditions

Now that the user is able to guess until they are correct, we need to check the guess to let them know whether they were too high or too low. The statement that allows us to do this is the `if` statement.

An `if` statement will select whether to run a block of statements once or not at all based on a condition, or a *conditional expression*.

We used a conditional expression before in the guessing loop: `(guess != theNumber)`. To check whether a guess is too high or too low, we just need a few more comparison operators: less than (`<`), greater than (`>`), and equal to (`==`).

First, instead of just telling the user what their guess was, let's write some code to check whether their guess was too low. Replace the last line of the `while` statement with the following two-line `if` statement:

```
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
            }
```

The `if` statement begins with the keyword `if`, followed by a conditional expression in parentheses. In this case, the condition is `guess < theNumber`, which

means the value of the user's guess is less than the value of the random secret number. Notice there's no semicolon after the parentheses, because the `println()` statement that follows is actually part of the `if` statement. The whole statement tells the program that if the condition is true, it should print the user's guess and let them know they guessed too low. We use the concatenation operator (+) between the user's guess and the string of text telling them the guess was too low. Note the space after the first double quote and before `is`. This separates the user's guess from the word `is`.

If you run the program now and enter a low guess, like **`1`**, the `if` statement should tell the program to say your guess is too low. That's a good start, but what if we guess a number that's too high instead? In that case, we need an `else` statement.

The `else` statement gives the program a way to choose an alternative path, or set of steps, if the condition in the `if` statement is not true. We can test for guesses that are too high or too low with an `if-else` statement pair. Let's add an `else` statement right after the `if` statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
❷ else if (guess > theNumber)
      System.out.println(guess + " is too high. Try again.");
```

Notice that the code at ❷ looks similar to the code at ❶. Often when we're using `if-else` statements, we need to check for multiple conditions in a row, instead of just one. Here, we need to check for a guess that's too low, too high, or just right. In cases like this, we can chain `if-else` conditions together by placing the next `if` statement inside the `else` portion of the previous `if-else` statement. At ❷ we've begun the next `if` statement immediately after the `else` from the previous condition. If the guess is higher than the number, the program tells the user their guess is too high. Now that the program can tell the user if their guess is too high or too low, we just need to tell them if they guessed correctly and won!

If neither of the previous conditions is true—the user's guess is not too high and not too low—then they must have guessed the number. So we add one final `else` statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
```

```
        ❷ else if (guess > theNumber)
              System.out.println(guess + " is too high. Try again.");
        ❸ else
              System.out.println(guess + " is correct. You win!");
```

Notice that we don't need a conditional expression for this final `else` statement ❸. A correct guess is the only remaining option if the number is neither too high nor too low. In the case of a winning guess, we provide the statement to let the user know they've won. The full program up to this point is shown in Listing 2-4. Save your *HiLo.java* file and run the program to check that it works. It should prompt you to enter guesses until you guess the correct number.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        // System.out.println( theNumber );
        int guess = 0;
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
            else if (guess > theNumber)
                System.out.println(guess + " is too high. Try again.");
            else
                System.out.println(guess + " is correct. You win!");
        }  // End of while loop for guessing
    }
 }
```

Listing 2-4: The Hi-Lo guessing game is complete for a single full round of play.

The full program is now a completely playable guessing game! After the user wins, the program tells them that they guessed correctly and won, and then it ends, as shown in Figure 2-6.

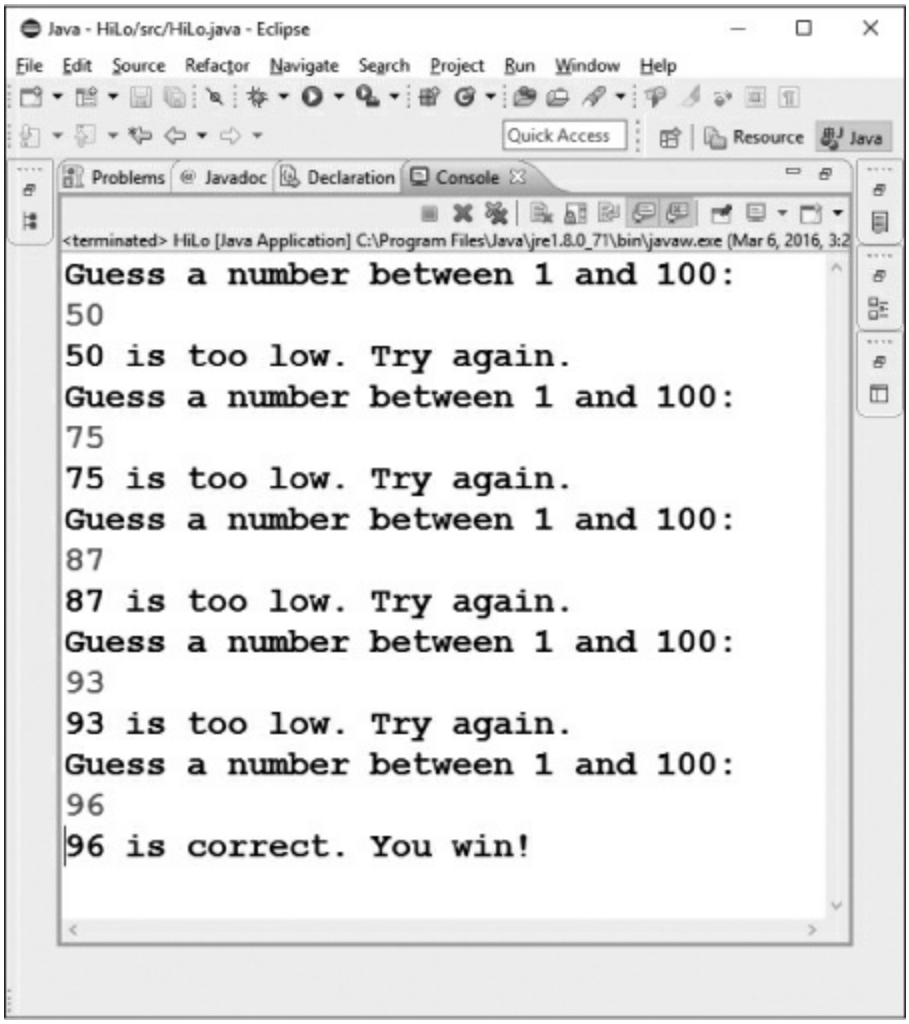

Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
Problems
Javadoc
Declaration
Console
<terminated> HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 6, 2016, 3:2
Guess a number between 1 and 100:
50
50 is too low. Try again.
Guess a number between 1 and 100:
75
75 is too low. Try again.
Guess a number between 1 and 100:
87
87 is too low. Try again.
Guess a number between 1 and 100:
93
93 is too low. Try again.
Guess a number between 1 and 100:
96
96 is correct. You win!

Figure 2-6: One full play-through of the Hi-Lo guessing game—the program ends when the user guesses the secret number.

Give yourself a hand! You've built a program in Java from scratch, and if this is your first program ever in Java, you deserve some kudos. Enjoy the game for a few rounds and see if you can guess the number in fewer tries each time. Test your program to make sure it works the way you want, and we'll add some improvements in the next section.

Adding a Play Again Loop

Right now, the only way to play the guessing game again is to rerun the program in Eclipse. Fortunately, we already know there's a way to make our program do something over and over again—we need another loop!

The guessing game program ends when the user guesses the right number because there's nothing after the `while` loop. The `while` loop ends when the condition `(guess != theNumber)` is no longer true. A user might want to play over

and over once they get the hang of the game. For this play again loop, we'll learn a new keyword and a new kind of loop: the `do-while` loop.

Like the `while` loop, a `do-while` loop repeats a block of statements as long as a condition is true. Unlike the `while` loop, however, the block of code inside a `do-while` loop is guaranteed to run at least once. There are times when the condition at the top of a `while` loop may be false before the loop even starts, so the entire loop and all the lines of code inside it are ignored. Think of the condition of a `while` loop as being like a thermostat on a heater. If the temperature of the room is already warm enough and the condition for the heater to turn on isn't met, the heater may not turn on at all.

For our guessing game, or almost any game program in general, we choose a `do-while` loop (we sometimes call this the *game loop*), because the user probably wants to play the game at least once. We also usually want to ask the user if they would like to play again, and the user typically responds yes or no (or `y` or `n` in a text-based game like this one). The game will continue to play through the game loop as long as the user responds with a yes.

To check the user's response, we'll need a new type of variable: a `String`. Strings are objects that hold text within double quotation marks, like `"y"`, or `"yes"`, or `"My name is Bryson! I hope you like my game!"`. Earlier we used an integer variable, or `int` type, to hold the numbers the user was guessing, but now we need to hold text, so we'll use a `String` instead. We can add a `String` variable to the top of the program, right after the `Scanner` setup:

```
Scanner scan = new Scanner(System.in);
String playAgain = "";
```

Notice the `String` type begins with an uppercase `S`. This is because the `String` type is actually a class, complete with several useful functions for working with strings of text. I've named the variable `playAgain`, using camel case with a capital `A` to start the second word. Remember, no spaces are allowed in variable names. And, just like how we gave an initial value of `0` to the `guess` variable with `int guess = 0`, here we've given an initial value to the `playAgain` variable with `playAgain = ""`. The two double quotes, with no space between, indicate an empty string, or a `String` variable with no text in it. We'll assign a different text value to the variable later, when the user enters `y` or `n`.

Just as we did with the `while` loop, we'll need to figure out which statements should be repeated in the `do-while` loop. The `do-while` loop will be our main loop,

so almost all of the statements in the program will go inside it. In fact, all the remaining statements after the `Scanner` and `String playAgain` statements will be contained in the `do-while` loop. Those steps describe one full round of play, so for each round, the game repeats all of those steps again, from choosing a new random number to declaring a winning guess and asking the user to play again.

So, we can add the `do` keyword and an opening brace immediately after these two lines and before the code that creates the secret number:

```
    Scanner scan = new Scanner(System.in);
    String playAgain = "";
    do {
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        // System.out.println( theNumber );
        int guess = 0;
        while (guess != theNumber) {
```

Then, after the closing brace for the `while` loop for guessing and the brace following our last `else` statement, we'll ask the user if they would like to play again and get their response from the keyboard.

Then we need to close the `do-while` loop with a `while` condition to check whether the user replied with a yes:

```
    }   // End of while loop for guessing
      ❶ System.out.println("Would you like to play again (y/n)?");
      ❷ playAgain = scan.next();
  ❸ } while (playAgain.equalsIgnoreCase("y"));
❹ }
❺ }
```

The prompt asks the user `"Would you like to play again (y/n)?"` ❶, to which they can reply with a single letter, `y` for yes or `n` for no. At ❷, the `scan.next()` function scans the keyboard for input, but instead of looking for the next integer as `nextInt()` does, it looks for the next character or group of characters that the user types on the keyboard. Whatever the user types will get stored in the variable `playAgain`.

The line at ❸ closes the block of code that repeats the game with a brace, and it contains the `while` condition that determines whether the code will run again. Within the `while` condition, you can see an example of the `equals()` method of a `String` object. The `equals()` method tells you whether a string variable is exactly

the same as another string of characters, and the `equalsIgnoreCase()` method tells you whether the strings are equal even if their capitalization is different. In our game, if the user wants to play again, they are asked to type `y`. However, if we just test for a lowercase `y`, we might miss an uppercase `Y` response. In this case, we want to be flexible by checking for the letter `y`, whether it is uppercase or lowercase, so we use the string method `equalsIgnoreCase()`.

The final `while` statement tells Java to continue to do the game loop while the string variable `playAgain` is either an uppercase or lowercase `y`. The final two closing braces at ❹ and ❺ are the ones that were already in the program. The one at ❹ closes the `main()` method, and the one at ❺ closes the entire `HiLo` class. I've included them just to show where lines ❶ through ❸ should be inserted.

The complete game to this point is shown in Listing 2-5.

```
import java.util.Scanner;
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
           // Create a random number for the user to guess
           int theNumber = (int)(Math.random() * 100 + 1);
           // System.out.println( theNumber );
           int guess = 0;
           while (guess != theNumber) {
               System.out.println("Guess a number between 1 and 100:");
               guess = scan.nextInt();
               if (guess < theNumber)
                   System.out.println(guess + " is too low. Try again.");
               else if (guess > theNumber)
                   System.out.println(guess + " is too high. Try again.");
               else
                   System.out.println(guess + " is correct. You win!");
         }   // End of while loop for guessing
         System.out.println("Would you like to play again (y/n)?");
         playAgain = scan.next();
      } while (playAgain.equalsIgnoreCase("y"));
    }
}
```

Listing 2-5: The Hi-Lo guessing game is ready to play over and over again.

Review your code to make sure you've added everything in the correct place, check your braces and semicolons, and take a moment to save your file. We'll test the game in the next section.

NOTE

Your indentation, which is the tab spacing at the beginning of each line, may not look exactly like the last code snippet because we've added braces in a couple of new places. Fortunately, adding new features, including loops and other blocks of code, is so common in Java that Eclipse has a menu option to clean up indentation automatically. First, select (highlight) all the text in your HiLo.java *file on the screen. Then, go to* ***Source ▸ Correct Indentation****. Eclipse will correctly indent each line of code to show which statements are meant to be grouped together. As I mentioned before, the indentation doesn't matter to the computer (the program will run just fine even with* no *tabs or extra spaces), but good indentation and spacing help make the program easier to read.*

Testing the Game

Now that the play again loop is in place, the game should run perfectly. First, save your *HiLo.java* file and choose **Run ▸ Run** to test the program. After you guess the first random number correctly, the program should ask you if you'd like to play again. As long as you respond **y** (or **Y**) and press ENTER, the program should keep giving you new random numbers to guess. In the screenshot in Figure 2-7, notice that the game starts over when I respond **y** to the prompt to play again.

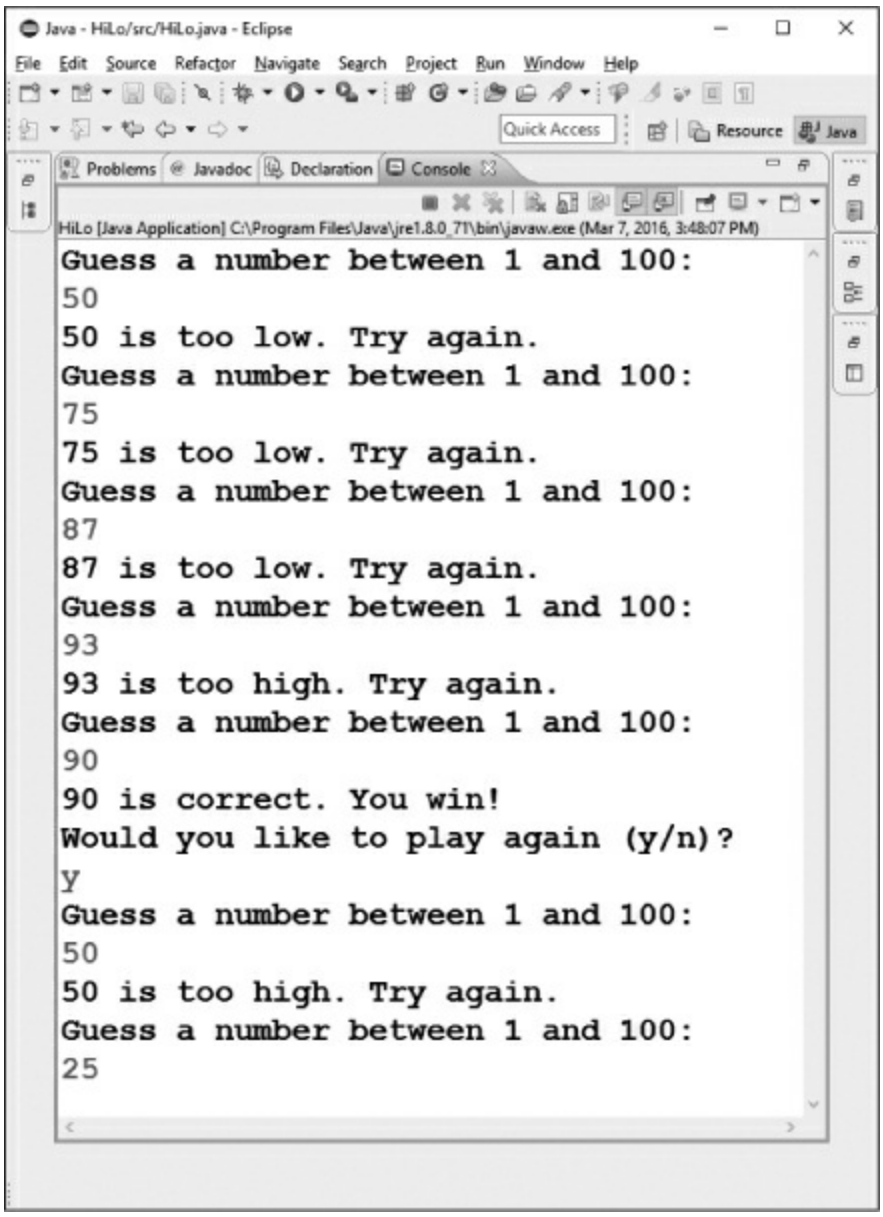

Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
Problems
Javadoc
Declaration
Console
HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 7, 2016, 3:48:07 PM)
Guess a number between 1 and 100:
50
50 is too low. Try again.
Guess a number between 1 and 100:
75
75 is too low. Try again.
Guess a number between 1 and 100:
87
87 is too low. Try again.
Guess a number between 1 and 100:
93
93 is too high. Try again.
Guess a number between 1 and 100:
90
90 is correct. You win!
Would you like to play again (y/n)?
y
Guess a number between 1 and 100:
50
50 is too high. Try again.
Guess a number between 1 and 100:
25

Figure 2-7: The guessing game is fully playable for multiple rounds as long as the user answers y or Y.

When the user finishes playing and responds to the play again question with `n`, or anything other than `y` or `Y`, the game will end. However, we might want to thank them for playing after they've finished the game. Add the following line after the final `while` statement, before the final two closing braces:

```
        } while (playAgain.equalsIgnoreCase("y"));
        System.out.println("Thank you for playing! Goodbye.");
    }
}
```

Finally, the last line we'll add to the guessing game app is to address a warning you may have noticed in Eclipse. This warning appears as a faint yellow line

under the declaration of the `scan` object, as well as a yellow triangle with an exclamation point to the left of that line. Eclipse is bringing to our attention that we've opened a resource that we haven't closed. In programming, this can create what's known as a *resource leak*. This doesn't usually matter if we just open one `Scanner` object for keyboard input, but if we leave multiple `Scanner` objects open without closing them, the program could fill up memory, slowing or even crashing the user's system. We use the `close()` method of the `Scanner` class to tell our program to close the connection to the keyboard.

Add the following line after the `println()` statement thanking the user for playing, before the final two closing braces:

```
        System.out.println("Thank you for playing! Goodbye.");
        scan.close();
      }
}
```

You'll notice that the yellow warning disappears from the Eclipse editor window when we add this line. Eclipse helps with common programming errors like misspellings or missing punctuation, and it even warns us about problems that *could* occur like resource leaks and unused variables. As you build bigger, more complex applications in Java, these features of the IDE will become even more valuable. You can find more information on using Eclipse to debug your programs in the appendix.

The finished program, shown in Listing 2-6, is a fully playable guessing game, complete with the option to play again and guess a new random number every game.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
           // Create a random number for the user to guess
           int theNumber = (int)(Math.random() * 100 + 1);
           // System.out.println( theNumber );
           int guess = 0;
           while (guess != theNumber) {
               System.out.println("Guess a number between 1 and 100:");
               guess = scan.nextInt();
               if (guess < theNumber)
                   System.out.println(guess + " is too low. Try again.");
```

```
            else if (guess > theNumber)
                System.out.println(guess + " is too high. Try again.");
            else
                System.out.println(guess + " is correct. You win!");
          }  // End of while loop for guessing
          System.out.println("Would you like to play again (y/n)?");
          playAgain = scan.next();
      } while (playAgain.equalsIgnoreCase("y"));
      System.out.println("Thank you for playing! Goodbye.");
      scan.close();
   }
}
```

Listing 2-6: The finished text-based, command line guessing game

There are a few things worth noting about the finished Hi-Lo guessing game program. First, despite all the work in writing it, the code is relatively short—fewer than 30 lines long. Nevertheless, you could play it forever if you wanted to. Second, this program not only demonstrates conditions and looping, it also makes use of a loop inside another loop. This is called a *nested loop*, because the guessing loop is contained in, or nested inside, the play again loop. The indentation helps us see where the `do-while` loop begins and ends, and we can see the smaller `while` loop and its `if` statements tabbed over and nested inside the bigger `do-while` loop. Finally, we end the program neatly—both for the user, by thanking them for playing, and for the computer, by closing the scanner resource.

What You Learned

While building a simple, fun, playable game, we've picked up several valuable programming concepts along the way. That's the way I first learned how to code as a kid—I would find a fun game or graphical app, program it, then change it, take it apart, and try new things. Play and exploration are an important part of learning anything new, and I hope you'll take a little time to try new things with each program. The programming challenges at the end of each chapter will also give you an opportunity to try a few new things.

In building this guessing game, we've developed a wide range of skills in Java:

- Creating a new class, `HiLo`
- Importing an existing Java package, `java.util.Scanner`

- Using a `Scanner` object to accept keyboard input
- Declaring and initializing integer and string variables
- Generating a random number with `Math.random()` and casting it to an integer
- Using `while` and `do-while` loops to repeat a set of steps while a condition is true
- Printing text strings and variable values to the command line console
- Scanning integers and strings from the keyboard and storing them in variables
- Testing various conditional expressions in `if` and `if-else` statements
- Using `String` methods to compare string values with `equalsIgnoreCase()`
- Closing input resources like `Scanner` objects with the `close()` method
- Running a command line program from inside Eclipse

In addition to practical skills, you've also developed a working knowledge of several important programming concepts in Java:

Variables `theNumber` is an integer variable, or `int`, and so is `guess`. `playAgain` is a string variable, or `String`. We change the values of these variables as we play the game by entering new number guesses or answering `y` or `n`.

Methods Methods are what we call functions inside a class in Java. `Math.random()` is a method for generating random numbers between 0.0 and 1.0. The `scan.nextInt()` method accepts numeric input from the user. `System.out.println()` is a function for displaying text to the console or terminal window.

Conditionals The `if-else` statements allow us to test whether a condition, like `guess < theNumber`, is true and run a different block of code depending on the outcome of that test. We also use conditional expressions to determine whether to perform a loop again, as in the statement `while (guess != theNumber)`. This statement will loop as long as `guess` is not equal to `theNumber`. Remember the test for "is equal to" is the double equal sign: `==`.

Loops A `while` loop lets us repeat a block of code as long as a condition is true. We used a `while` loop in the guessing game to keep asking the

user for another guess until they got the right number. A `do-while` loop always runs at least once, and we used one to ask the user if they wanted to play again.

Classes The whole HiLo app is a Java class, `public class HiLo`. A class is a template. Now that we've built a class template for the HiLo guessing game, we can reuse it to play the guessing game across many different computers. We also imported the `Scanner` and `Math` classes in this app to accept user input and generate random numbers. We'll write our own classes to do something new, and we'll take advantage of the classes already included in Java to do everyday tasks like input, math, and more.

Programming Challenges

Try these programming challenges to review and practice what you've learned and to expand your programming skills. If you get stuck, you can visit the book's website at *https://www.nostarch.com/learnjava/* to download sample solutions, or you can watch this lesson in the video course online at *http://www.udemy.com/java-the-easy-way/* for step-by-step solutions. Chapter 2 is free to preview, and you can use the coupon code BOOKHALFOFF to save 50 percent when you buy the full course.

#1: Expanding Your Range

For this first programming challenge, change the guessing game to use a bigger range of numbers. Instead of 1 to 100, try having the user guess between –100 and 100!

HINT

Multiply Math.random() by 200 and subtract 100 from the result.

Remember to change *both* the programming statement that generates the random number *and* the prompt that tells the user the range they should guess between.

If you want an easier game, you can change the range from 1 to 10 and wow your friends when you can guess the secret number in just four tries. Try other ranges, like 1 to 1,000, or even 1 to 1,000,000, or use negative ranges if you want! (Remember that you can't use commas when writing the number in Java.) You'll not only get better at programming but also improve your math skills. Change the program however you'd like and have fun with it!

STRATEGIZE YOUR GUESSES

You may discover that the more you play the guessing game, the faster you'll be able to guess the secret number. You might even stumble onto the fact that you can guess the number fastest by guessing in the middle of a range with each new guess. This technique is called a *binary search*. Guessing a number in the middle of the possible range cuts the number of possibilities in half each time.

Here's how it works. For a number from 1 to 100, guess 50. If that's too low, you know the secret number must be between 51 and 100, so guess 75, right in the middle of this range. If that's too low, try a number halfway between 76 and 100, which would be 87. One reason the binary search is so valuable is that we can reduce the number of guesses to just seven tries (or fewer) to find a secret number between 1 and 100, every time. Try it out!

When you get the hang of guessing a number between 1 and 100 in seven tries or less, try guessing a number from 1 to 1,000 in just 10 tries. If you're really brave (and have a pencil nearby), try guessing a number from 1 to 1,000,000. Believe it or not, it should take you just 20 guesses.

#2: Counting Tries

We've already built a pretty cool guessing game app, but let's try adding one more feature to the game. Your challenge is to count and report how many tries it takes the user to guess the secret number. It could look something like the following:

```
62 is correct! You win!
It only took you 7 tries! Good work!
```

To accomplish this task, you'll need to create a new variable (you might add a line like `int numberOfTries = 0;`), and you'll have to add to the number of tries every time the guessing loop executes. You can do this by increasing the variable `numberOfTries` by one for each new loop using `numberOfTries = numberOfTries + 1`. Be sure to include text to let the user know the number of tries.

It may take a few tries to get all the code working in the right order at the right time, but it's worth the effort and will help you practice your new skills. In Chapter 3, we'll build this feature into a different version of the guessing game. In the meantime, I hope you'll come up with even more ideas for improving and changing the game. Playing with your programs, taking them apart, and rebuilding them can be the best way to learn.

#3: Playing MadLibs

For your final challenge in this chapter, let's write a completely new program. We've learned how to ask a user for input and store it in a variable. We've also learned how to print out both text and variable values to the screen. With those skills, you can build even more interesting and fun programs.

Have you ever played MadLibs? Let's try to use our new skills to build a program in that same style. MadLibs asks a player for various words or parts of speech, such as a color, a past-tense verb, or an adjective, and then inserts the words the player chose into a template, usually resulting in a funny story. For example, if a player gave a color of "pink," a past-tense verb of "burped," and an adjective of "silly" and then inserted them into the template "The ____ dragon ____ at the ____ knight," they would get the result "The *pink* dragon *burped* at the *silly* knight."

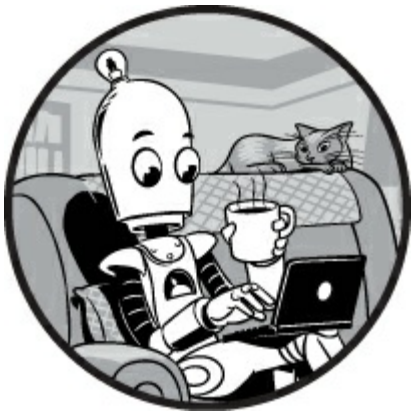
Now, the challenge is to write a new program, *MadLibs.java*, with a class called `MadLibs` and a `main()` method that prompts the user for several words. Those words should each be stored in a different `String` variable, like `color`, `pastTenseVerb`, `adjective`, and `noun`, which you initialize as empty strings. Then, after the user has entered their last word, the program should print a completed sentence or story by replacing the empty strings with the words the user provided, like this:

```
System.out.print("The " + color + " dragon " + pastTenseVerb + " at the " +
adjective);
System.out.println(" knight, who rode in on a sturdy, giant " + noun + ".");
```

Note that the first statement is a `print()` statement instead of a `println()`. The `print()` statement continues printing at the end of the same line, allowing us to build a longer paragraph or story. The `println()` statement, however, always skips a line after printing, like when you press ENTER at the end of the line. You can write a longer MadLibs story by using different variable names like `noun1`, `noun2`, and `noun3`. Give it a try, and get ready to laugh at the funny stories you create! Try to personalize each program you create by adding new features and making it your own.

3

CREATING A GUI FOR OUR GUESSING GAME



In this chapter, we'll create a GUI version of the Hi-Lo guessing game from Chapter 2, as shown in Figure 3-1. When this version of the program runs, it presents a graphical, or visual, interface that users will interact with. The GUI version allows the user to play the game in a desktop window like the applications you use every day. It's a professional, windowed app complete with a text field for the user's guess, a button to submit the guess, and a label that tells the user if they've guessed too high, too low, or correctly.

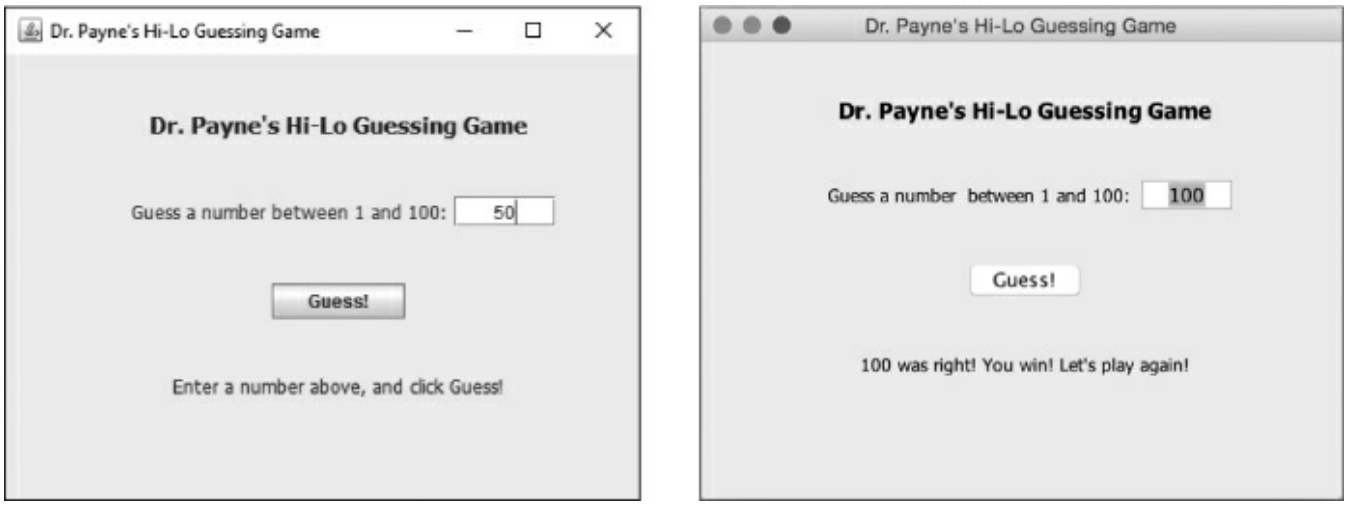

Dr. Payne's Hi-Lo Guessing Game
Dr. Payne's Hi-Lo Guessing Game
Guess a number between 1 and 100: 50
Guess!
Enter a number above, and click Guess!
Dr. Payne's Hi-Lo Guessing Game
Dr. Payne's Hi-Lo Guessing Game
Guess a number between 1 and 100: 100
Guess!
100 was right! You win! Let's play again!

Figure 3-1: A GUI version of the Hi-Lo guessing game running on Windows (left) and on macOS (right)

Best of all, the exact same Java source code produced both versions of the app

in Figure 3-1. With the GUI code we'll write in this chapter, our game will be playable on Windows, macOS, and Linux!

Practicing with JShell

JShell works at the command line, accepting text commands and usually responding with text output. However, JShell also has access to the full set of Java libraries—prewritten packages of code—and isn't limited to just text. Before we start programming our game, let's create a simple GUI in JShell as practice.

Creating a GUI in Four Lines of Code

We can create a simple GUI window with just four lines of code in JShell. Let's take each statement one at a time. First, enter an `import` statement into JShell to import the `javax.swing.JFrame` class:

```
jshell> import javax.swing.JFrame
```

The `JFrame` class we've just imported creates a frame or window using Java. Next, let's create a frame using that class. Enter the following declaration to create a `JFrame` called `myFrame`:

```
jshell> JFrame myFrame = new JFrame("Hello!")
```

The `new` keyword creates a new object of the `JFrame` class—in this case, a GUI window that we'll program to say `"Hello!"` in its title bar. JShell responds with a long line letting us know some of the default properties the window will have:

```
myFrame ==>
javax.swing.JFrame[frame1,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayou
...
```

The information shown after the square bracket is a string representation of `myFrame` and its property values, like its size (`0x0`, which is 0 by 0 pixels) and whether it's hidden or visible. Let's change one of those properties by setting the size to something bigger than 0 by 0 pixels.

The third line of code sets the size of the window in pixels by specifying a width and height for the frame:

```
jshell> myFrame.setSize(300,200)
```

This tells Java to make the window 300 pixels wide and 200 pixels tall, big enough for us to see the window and title bar.

Finally, let's show the window on the screen by calling the `setVisible()` method:

```
jshell> myFrame.setVisible(true)
```

We use `setVisible(true)` to show a window and `setVisible(false)` to hide it. Since we've called `myFrame.setVisible(true)`, you should see your window appear on the screen, as shown in Figure 3-2.

Figure 3-2: Our GUI window on Windows (left) and on macOS (right)

We've created a GUI window by entering just four snippets of code into JShell!

Creating an Interactive GUI in 10 Lines of Code!

I showed the previous example to my students at the university and to my sons at home, and both groups responded with something like "That's really cool, but can you make the window *do* something?"

Thankfully, the answer is yes. By adding just a few more lines of code, we can add a button to the window that prints something every time you click it.

Let's begin with a fresh JShell. Clear your JShell history by entering **`/reset`** at the JShell prompt:

```
jshell> /reset
|  Resetting state.
```

Next, let's use the built-in JShell editor to write a 10-line snippet of code to create an interactive GUI app that responds to your click. At the JShell prompt,

enter **/edit** to open the JShell editor:

```
jshell> /edit
```

The JShell Edit Pad, which is helpful when we want to write several lines of code at once or when we want to go back and edit a line we've typed earlier, appears with an empty window as shown in Figure 3-3.

Figure 3-3: The JShell Edit Pad is a convenient editor for writing longer snippets of code.

In this case, we want to type 10 lines of code that will create an interactive GUI window with a button that prints something when it's clicked. Enter the following lines, being extra careful with your capitalization and adding a semicolon at the end of each complete statement to separate the multiple commands:

```
  import javax.swing.*;
  JFrame window = new JFrame("Bryson's Window");
❶ JPanel panel = new JPanel();
  JButton button = new JButton(❷"Click me!");
❸ panel.add(button);
❹ window.add(panel);
  window.setSize(❺300,100);
❻ button.addActionListener(e ->
       System.out.println("Ouch! You clicked me!"));
❼ window.setVisible(true);
```

First, we import all the Swing GUI classes, including `JFrame`, `JPanel`, and `JButton`. An asterisk (`*`) at the end of a Java library is called a *wildcard character* and means "include every class in this package." Then, we create a `JFrame` just like we did in the previous example. At ❶, we create a panel inside the window. The panel will serve as a container for other GUI components like labels and buttons.

Next, we add a button with the text "Click me!" ❷ printed on it. At ❸, we add the button to our GUI panel, and then we add the panel to the window at ❹. Then, we set the size of the window to 300 pixels wide by 100 pixels high ❺, similar to the previous example.

The lines at ❻ are where the magic happens: here we add an action listener to the "Click me!" button to respond every time the user clicks the button. The action listener will print `"Ouch! You clicked me!"` to the console whenever the user clicks the GUI button. We'll use listeners like this one in the GUI apps we build in this book to make the programs interact with the user whenever the user performs an action.

Finally, we make the window visible ❼, and we're ready to test the program by clicking the button.

When you've finished typing all 10 lines of code and double-checked them for accuracy, click the **Accept** button in JShell Edit Pad to accept the code and run it in JShell, as shown in Figure 3-4.

After accepting the code, click **Exit** to close the JShell Edit Pad. JShell will run the code snippet, and if you've typed everything correctly, you'll see a small window pop up like the one in Figure 3-5.

*Figure 3-4: After typing all 10 lines of code into the JShell Edit Pad, click **Accept**.*

Figure 3-5: An interactive GUI window with a clickable button

Click the button labeled **Click me!** with your mouse, and you'll see Java respond as follows in the JShell window:

```
jshell> Ouch! You clicked me!
Ouch! You clicked me!
Ouch! You clicked me!
Ouch! You clicked me!
```

If you close the small GUI window, all you have to do to get it back is repeat the last line that sets the window to visible. Press ENTER a couple of times after clicking back into the JShell command line window to get the prompt and cursor back and then enter the following:

```
jshell> window.setVisible(true)
```

The window reappears when we set the visible property to `true`; if you close the window, the visible property becomes `false`.

This 10-line snippet is cool enough, but now you'll learn how to build a game with a GUI!

Setting Up the GUI App in Eclipse

If you still have the *HiLo* project open in Eclipse from Chapter 2, as in Figure 3-6, close or minimize the editor window for any other Java files you may have been working on ❶. Also, collapse the *HiLo* project by clicking the small down arrow next to the *HiLo* folder in the Package Explorer on the left side ❷. This allows us to keep projects together in a single workspace without getting files mixed up.

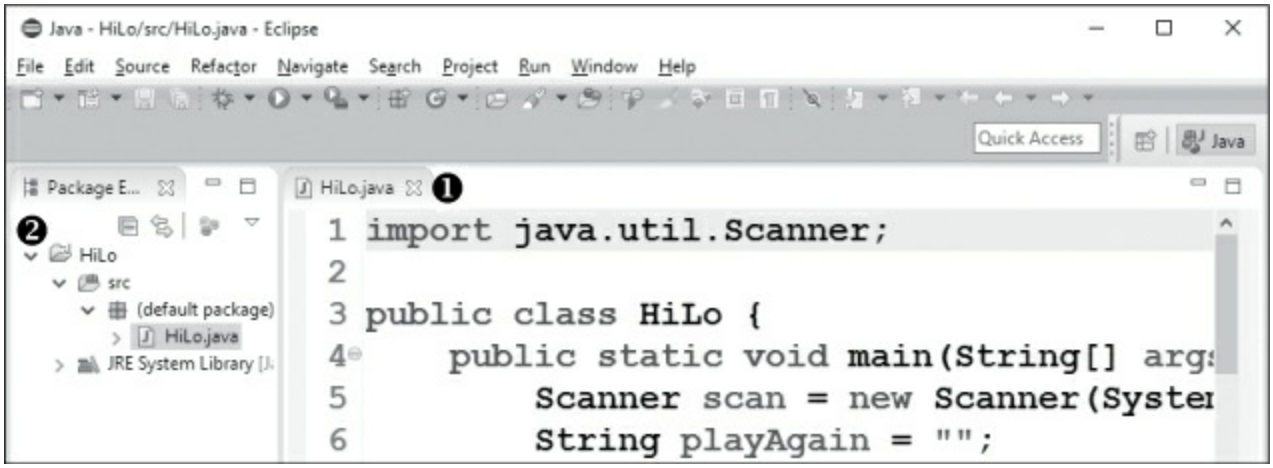


Figure 3-6: Close any open files and collapse the HiLo *project folder.*

From the Eclipse menu bar, go to **File ▸ New ▸ Java Project** and name the project *GuessingGame*. After you've typed in the project name, click the **Finish** button at the lower right. This creates the new project folder in which we'll develop the GUI version of the guessing game.

Expand the *GuessingGame* folder in the Package Explorer window and find the *src* folder. Right-click (or, on macOS, CONTROL-click) the *src* folder and select **New ▸ Class**. Name the new class `GuessingGame`. Be sure to use camel case, beginning with an uppercase *G* for the class name. Check the box next to the **public static void main(String[] args)** method stub. This adds the skeleton of a `main()` method so that we can run our app as a stand-alone program.

Before we finish, we'll need to do one more step differently than in the command line app we built in Chapter 2. We're going to change the superclass from the default `java.lang.Object` to the `javax.swing.JFrame` class, like the `JFrame` we used in JShell at the beginning of this chapter. A *superclass* or *parent class* in Java is a class that we extend in order to reuse code that's already been written, which in this case is the code needed to build a graphical, windowed interface. The `JFrame` class in the `javax.swing` package is one way we can include GUI components in our own apps, so we'll use `JFrame` to create a window and then customize a version of it to add some more features. Since the `JFrame` class is an extension of the `Object` superclass itself, we don't need `java.lang.Object` because `JFrame` will *inherit* the `Object` superclass's code. That means that `JFrame` will have all the features of the `Object` class, plus some. We'll see how extensions work in a bit.

Figure 3-7 shows these settings in the New Java Class dialog for our guessing game.

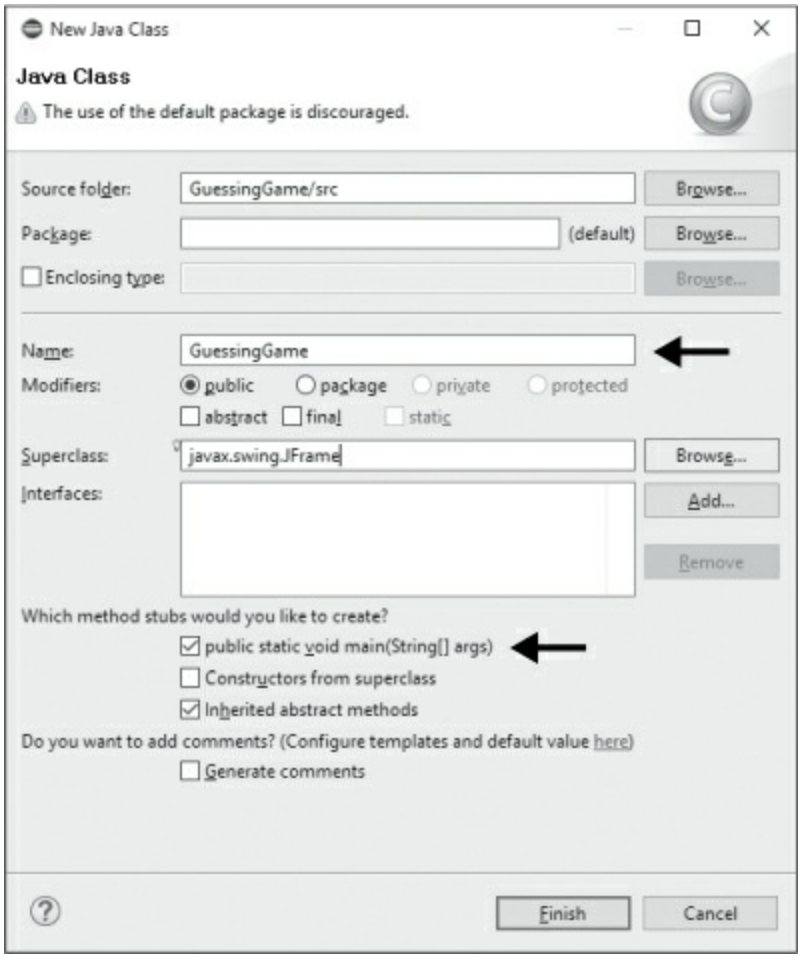


Figure 3-7: Be sure to change the superclass to javax.swing.JFrame to prepare for a GUI class.

Click **Finish** to see the following source code for *GuessingGame.java*:

```
❶ import javax.swing.JFrame;
   public class GuessingGame ❷extends JFrame {
       public static void main(String[] args) {
           // TODO Auto-generated method stub
       }
}
```

On the first line, Eclipse has imported the `javax.swing.JFrame` class ❶ to allow our program to use the GUI capabilities of Swing's set of packages. In the second line, we use a new coding superpower, the `extends` keyword ❷.

In object-oriented programming, a parent class or superclass can have *child classes* that inherit all the methods and attributes contained in the parent class. Once we write a class that does something important that we plan to reuse, we can refer to that original class and *extend* it to add new functionality without changing the code in the parent class. In this case, a `JFrame` parent class has the

ability to display text fields, labels, buttons, and other GUI components that we can customize and arrange for the new guessing game app.

The `JFrame` parent class we're extending in this app will enable us to display a window with GUI elements like text fields, labels, and buttons by designing the GUI layout with the WindowBuilder Editor in Eclipse. Let's see how the WindowBuilder Editor works next.

GUI Design with Eclipse's WindowBuilder Editor

The most popular Java IDEs include tools to help programmers build an attractive GUI in a design view, with a what-you-see-is-what-you-get (WYSIWYG) interface. WYSIWYG interfaces allow users to position design elements as they will look in the final product. For instance, Microsoft Word is a WYSIWYG interface, because it allows you to edit text as it will look in its final print form. In the same way, Java IDE tools like the WindowBuilder Editor allow programmers to place GUI components like text fields, labels, and buttons as they will look in the final windowed application, helping programmers deliver professional-looking GUI apps.

To open the WindowBuilder Editor, right-click the *GuessingGame.java* file in the Package Explorer on the left and then select **Open With ▸ WindowBuilder Editor**. A new window that looks like a regular text editor will open. But, if you look at the lower-left corner of the window, you'll see two new tabs, Source and Design. The Source tab is the plain text, source code view of your application. If you click the Design tab, you'll see the design view shown in Figure 3-8.

The WindowBuilder Editor appears only for GUI classes, such as `GuessingGame`, that extend `JFrame` or one of the other GUI superclasses. Notice that the preview window has a title bar with a Java icon on the left side and three buttons to minimize, maximize, and close the window. The preview window in Figure 3-8 looks slightly different on Windows, macOS, and Linux. This window will be our playground for building a GUI guessing game that looks and feels like a real app.

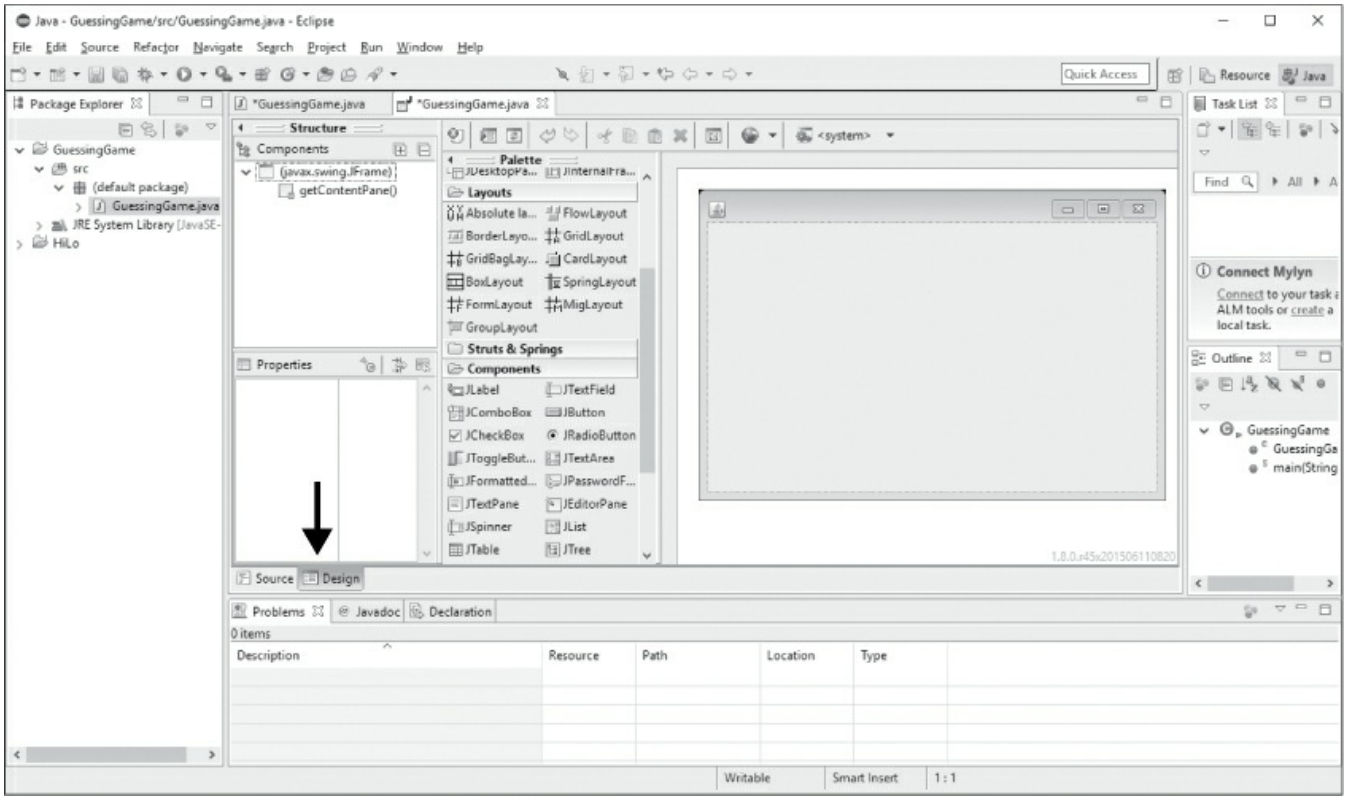


Figure 3-8: The WindowBuilder Editor's Design tab makes it easy to build GUI windowed apps.

Designing the User Interface

Double-click the *GuessingGame.java* tab at the top of the WindowBuilder Editor to expand the WindowBuilder design view to fullscreen. This will create more room for designing the GUI layout. You can double-click the tab again to restore the normal Java perspective when you're ready to get back to programming.

Setting GUI Properties in the Properties Pane

We'll use the Properties pane to start customizing the GUI window for our guessing game. The Properties pane can be found on the bottom left of the Design tab in the WindowBuilder Editor. As you can see in Figure 3-9, if you click a component in the Components pane (on the left under Structure), the Properties pane immediately below it lists several properties and values we can view and edit for that component.

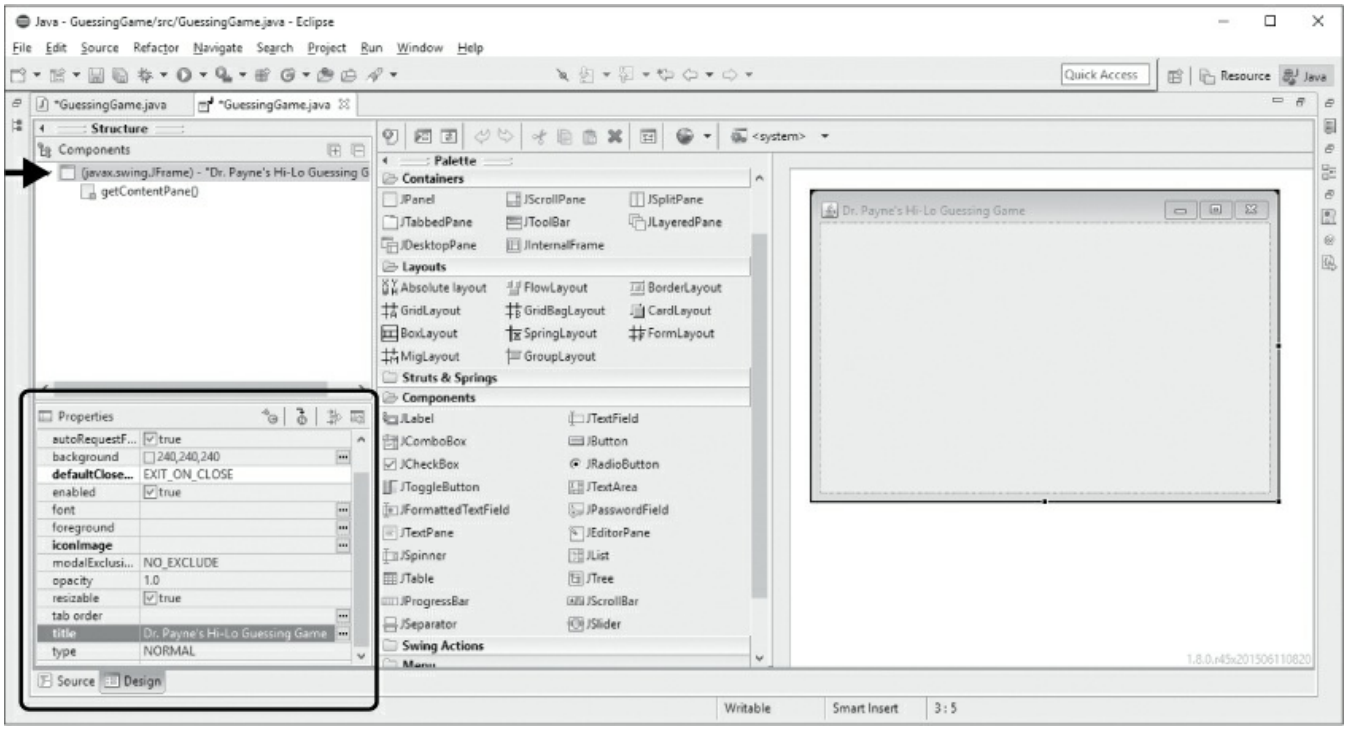


Figure 3-9: The Properties pane helps you customize the properties of each GUI component in your app.

We're going to build a customized GUI window for the guessing game app. First, let's change the title property of this `JFrame`. In the Components pane, click **javax.swing.JFrame**, then scroll down in the Properties pane until you see the `title` property. Click in the empty box to the right of `title` and enter ***`Your Name`*`'s Hi-Lo Guessing Game`**. (Replace the italicized text with your name; this is your app, after all!) After you enter the title text, press ENTER. The GUI preview on the right will automatically update to show *Your Name*'s Hi-Lo Guessing Game in the title bar.

With the title done, let's set a few other properties to make the GUI look and behave like we want for the guessing game. Scroll up in the Properties pane to find `defaultCloseOperation` (it may just display part of the name like `defaultClose...` or something similar if the pane is too narrow). Click the box to the right of `defaultCloseOperation` and select **EXIT_ON _CLOSE** from the drop-down menu. This means that when you click the close window button on the title bar of this `JFrame`, you'll exit the program. This is usually the default, but there might be times you want to close a window but not close the whole application—for example, in a save dialog or a pop-up window. For this single-window game, though, we want to exit the application when the main window closes.

Next, we'll change the way GUI components are arranged inside the app

window. Click the **getContentPane()** entry in the Components pane, just under `javax.swing.JFrame`. A *content pane* is the *interior* of the `JFrame`, where we'll build the guessing game GUI layout. Now, find Layout in the Properties pane and click the down arrow in the right edge of the field. Select **Absolute Layout** from the drop-down list of values; this enables us to place GUI elements with pixel-perfect precision.

Customizing GUI Components in the Palette Pane

Let's start customizing our guessing game app. You can find the elements you'll need for almost any GUI app in the Palette pane. The Palette pane can be found in the middle of the WindowBuilder Editor on the Design tab. Note that you can expand or collapse a pane by clicking the small arrow to the left of the pane's title. Click the arrow once to collapse the pane. Click the arrow again to expand the pane. This can be handy when you need extra room for a large, complex GUI layout.

In the Palette, scroll down until you see the Components section, as shown in Figure 3-10.

The Palette contains all the standard components you're probably already familiar with, like labels (`JLabel`), text fields (`JTextField`), buttons (`JButton`), and checkboxes (`JCheckBox`). You might have noticed that GUI components in the `javax.swing` package all begin with an uppercase J, followed by the name of the component in camel case. This makes it easy to remember the class name for each of the GUI elements in an app that uses the Swing toolkit.

Let's place a label at the top of the GUI window that says *Your Name*'s Hi-Lo Guessing Game. Click **JLabel** under Components in the Palette. Then, mouse over the GUI preview on the right; you should see gridlines appear. By default, the WindowBuilder Editor provides these gridlines to help you place elements. Now, hover your mouse near the top center of the gray content pane (the interior of the `JFrame`) and click to place a `JLabel` in that spot.

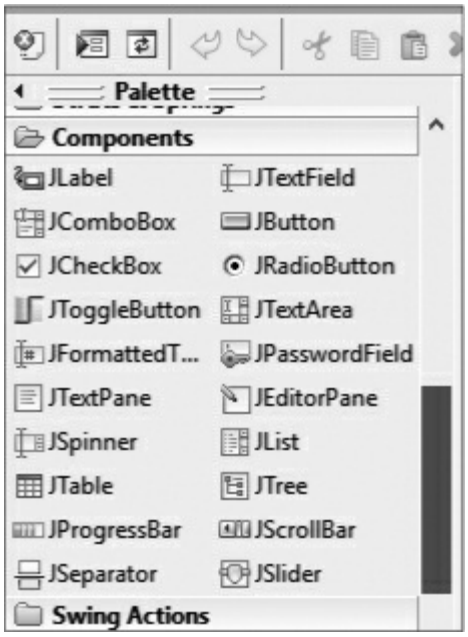


Figure 3-10: The Components section in the Palette pane is where we'll find the most common GUI components.

When you first place the `JLabel`, you can edit the text in the label directly in the GUI preview. Type ***`Your Name`*`'s Hi-Lo Guessing Game`** and press ENTER. If you want to change the text later, go to the `text` property of the `JLabel` in the Properties pane on the lower left and type in your text. At first, you'll probably only see part of the text in the label, because the label is too small to show the full text. Click the label, and you'll see small black resizing squares on each corner. Click the lower-left corner of the label and drag it to the left border of the content pane to stretch the label to the content pane's left edge. Then, click the label's lower-right corner and drag to resize it to the content pane's right edge.

Now your full label text should fit inside the label at the top of the GUI preview. Next let's make the label text centered and bold. First, find the `horizontalAlignment` property of the label in the Properties pane. Click the value and choose **CENTER** from the drop-down list. Next, find the `font` property of the label and click the three dots to the right of the `font` property's value. A font chooser window will open, allowing you to choose a font, style, and size. I chose Tahoma, 15 point, and Bold, as shown in Figure 3-11. Note that you may have to resize your label to fit a larger font.

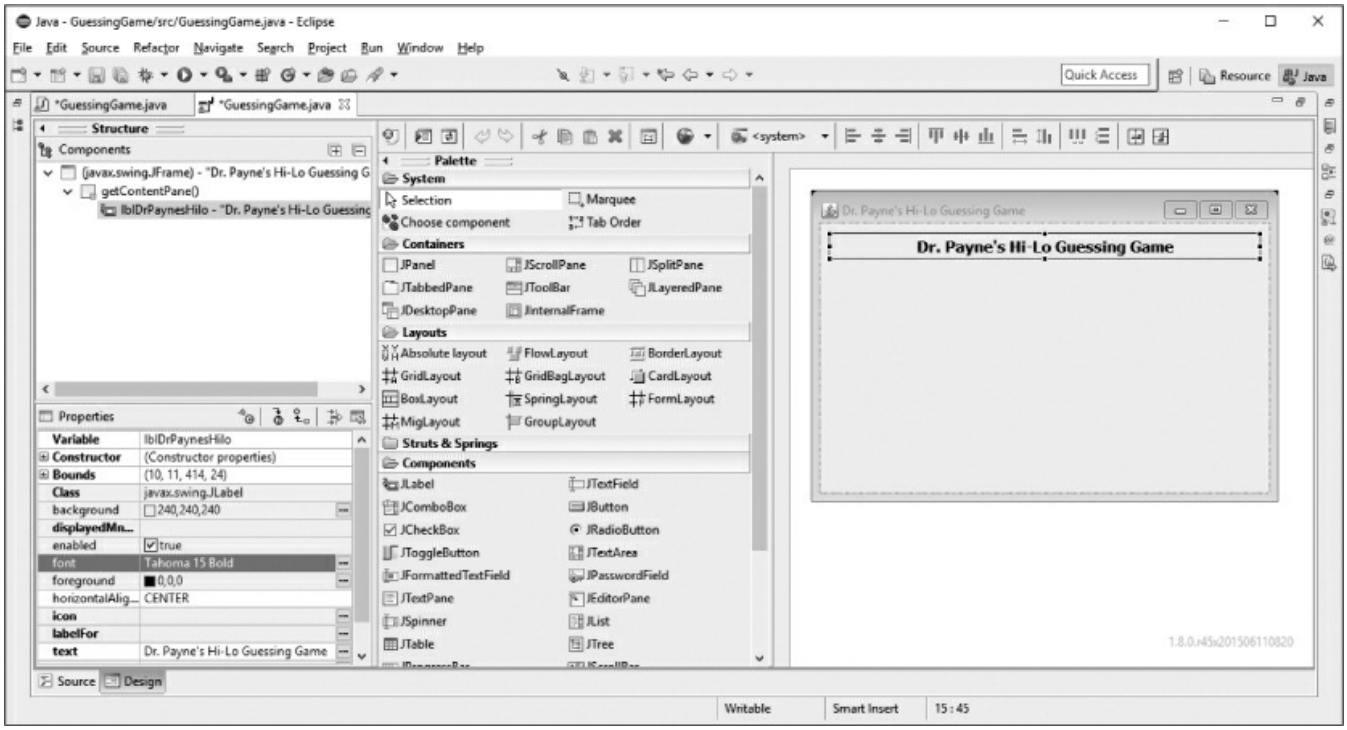


Figure 3-11: The first label, customized and in place, at the top of the content pane

Let's add a label for the first prompt a user will see in the game. Click **JLabel** in the Palette pane, then place the new label just above the middle of the content pane. Enter **`Guess a number between 1 and 100:`** as the text of the label. You'll need to resize the label a little wider than the text for extra padding.

Just to the right of the label, let's place a text field for the user to enter their guess. Click **JTextField** on the Palette and then drop the text field into place to the right of the label you just placed.

Resize it so it's just big enough to hold a three-digit number. Click the label again and change its `horizontalAlignment` property to **RIGHT** to bring the text closer to the text field, as shown in Figure 3-12.

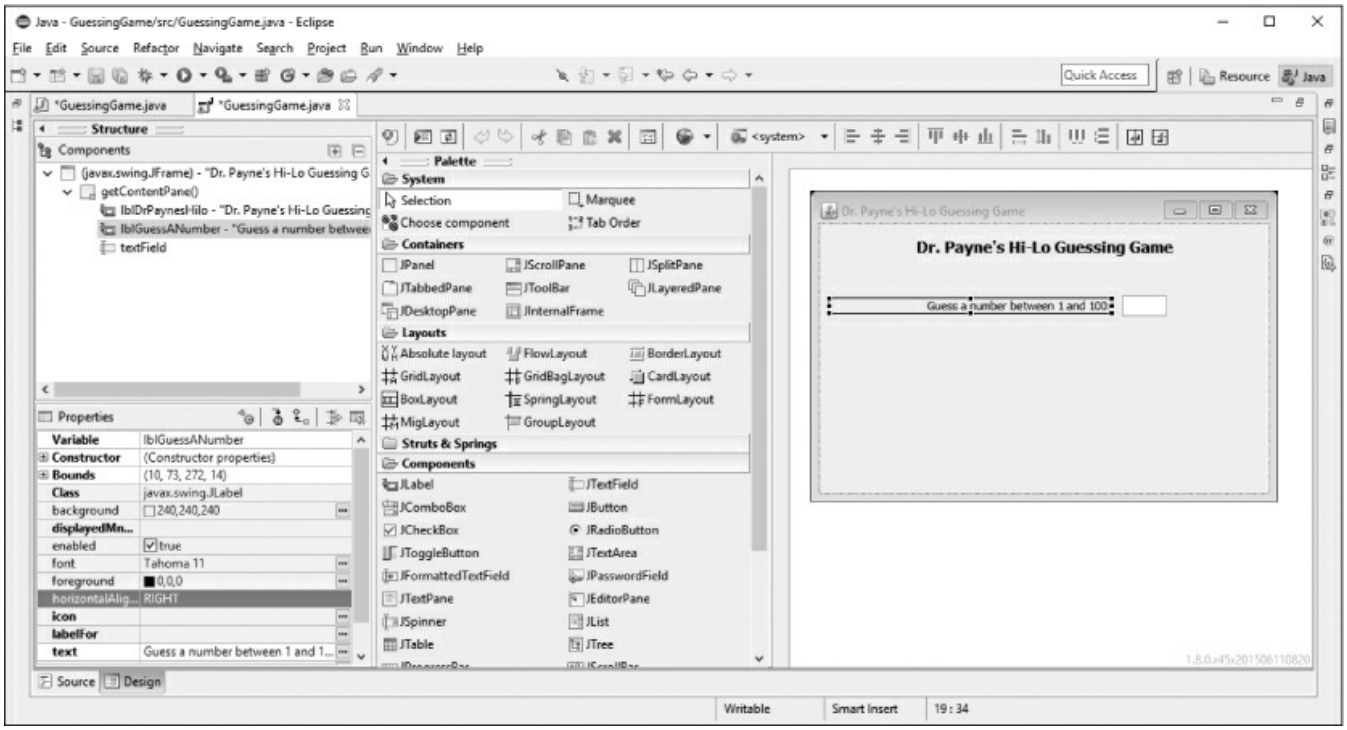


Figure 3-12: The guessing game now has a label and text field for the user's guess.

Next, let's place a button the user can push to submit their guess. On the Palette, find and click **JButton**. Mouse over the middle of the GUI preview window and click to place the `JButton`. Change the text to **`Guess!`**.

Finally, place a `JLabel` below the button, give it the text **`Enter a number above and click Guess!`**, and resize the label to the full width of the `JFrame`. Then, change the `horizontalAlignment` to **CENTER**. Later, this label will also be where we tell users whether they guessed too high, too low, or correctly. The alignment of each component may not be perfect yet, but your GUI layout should look something like Figure 3-13.

Dr. Payne's Hi-Lo Guessing Game
Dr. Payne's Hi-Lo Guessing Game
Guess a number between 1 and 100:
Guess!
Enter a number above and click Guess!

Figure 3-13: The GUI layout has all the components, just not perfectly aligned yet.

Now that we have all the GUI components on the screen, we're ready to

make a few tweaks to balance and center the layout. First, be sure to save your file.

Aligning GUI Elements

So far, we've been positioning GUI components by eye, trying to space them correctly and center them as much as possible. However, when we run a program, we expect the layout to be "perfect." Fortunately, Eclipse has built-in tools to help us align components.

First, let's distribute the four main elements evenly, spacing them the same distance apart vertically. Select the three labels and the button by clicking each one while holding down the SHIFT button. For now, don't click the text field where the user enters their guess. You should see a small row of alignment tools appear above the GUI preview window. Mouse over each tool to see a tool tip that tells you what that button does. You may need to resize your Eclipse window if you don't see all the tools at first.

Click the tool that looks like stacked buttons (the rightmost icon in Figure 3-14). All four selected items should now be spaced an equal distance apart from top to bottom.

Now select just the text field. Move it to realign it with the label prompting the user to guess a number. We didn't include the text field originally because Eclipse would have distributed all five components evenly, separating the text field and its label from each other and creating five rows instead of four.

Figure 3-14: You can align and space components evenly using the tools above the GUI preview window.

NOTE

If you ever mess up while aligning things (or doing anything else) in WindowBuilder, you can undo your previous action by pressing CTRL-Z or ⌘-Z. This is a great safety feature in the WindowBuilder Editor and lets you try new things without permanently messing up the layout.

Finally, click the button labeled **Guess!** and click the alignment button above the GUI preview (the one near the far right with the tool tip "Center

horizontally in window"). You can center or adjust the other components as desired. If you want to move two or more components together, you can SHIFT-click to select both, then click and drag to move them.

When you're happy with the layout, it's time to prepare for coding by naming the components. Take a moment to save your changes.

Naming GUI Components for Coding

The user interface of the guessing game app is ready for the player. To make the app easier to work with, however, we need to make a few final tweaks over in the Java source code file, where we'll be programming the rest of the game. We need to name each component so that we'll know how to refer to it inside the source code.

This step is sometimes called "wiring up," because we'll be connecting each GUI component needed in the program to variable names that we can access in the Java source code. Eclipse has been naming each GUI element for us as we've added them, but we'll want to change those names. Select the text field where the user will enter their guess. If you look in the Properties pane, you'll see the very top property is `Variable`. The variable name Eclipse has given your first text field by default is probably something like `textField`. Click the value field beside the `Variable` property and rename the variable for this text field to **`txtGuess`**, as shown in Figure 3-15.

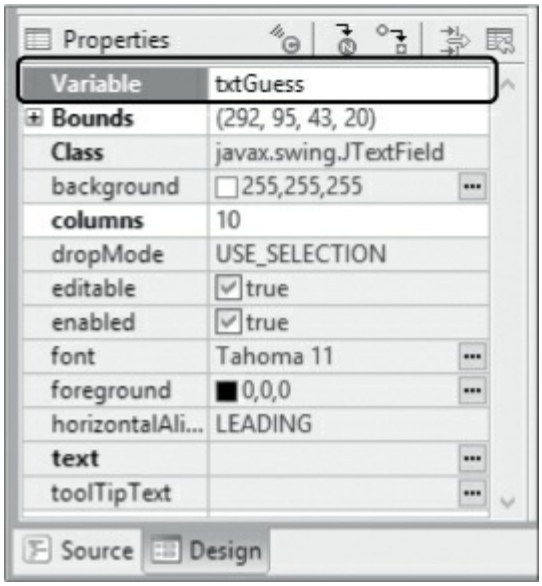


Figure 3-15: Rename the variables for each GUI element so we can use them easily in the Java source code.

The name `txtGuess` will remind us that it's a text field and that it stores the user's guess. You'll find that naming GUI components consistently and logically will help you code more quickly and efficiently, with fewer errors.

After you've renamed the text field, select the label at the bottom of the GUI window that reads "Enter a number above and click Guess!". This is the label that will output information for the user to read, like `Too high`, `Too low`, or `You win!`, so let's rename the variable **`lblOutput`**. This name will help us remember it's a GUI label (`lbl`) and that it's the one we want to use for output to display information to the user.

In the Source tab at the lower left of the WindowBuilder Editor window, you'll see that Eclipse has been writing the Java source code to produce the GUI layout you've been designing (Figure 3-16).

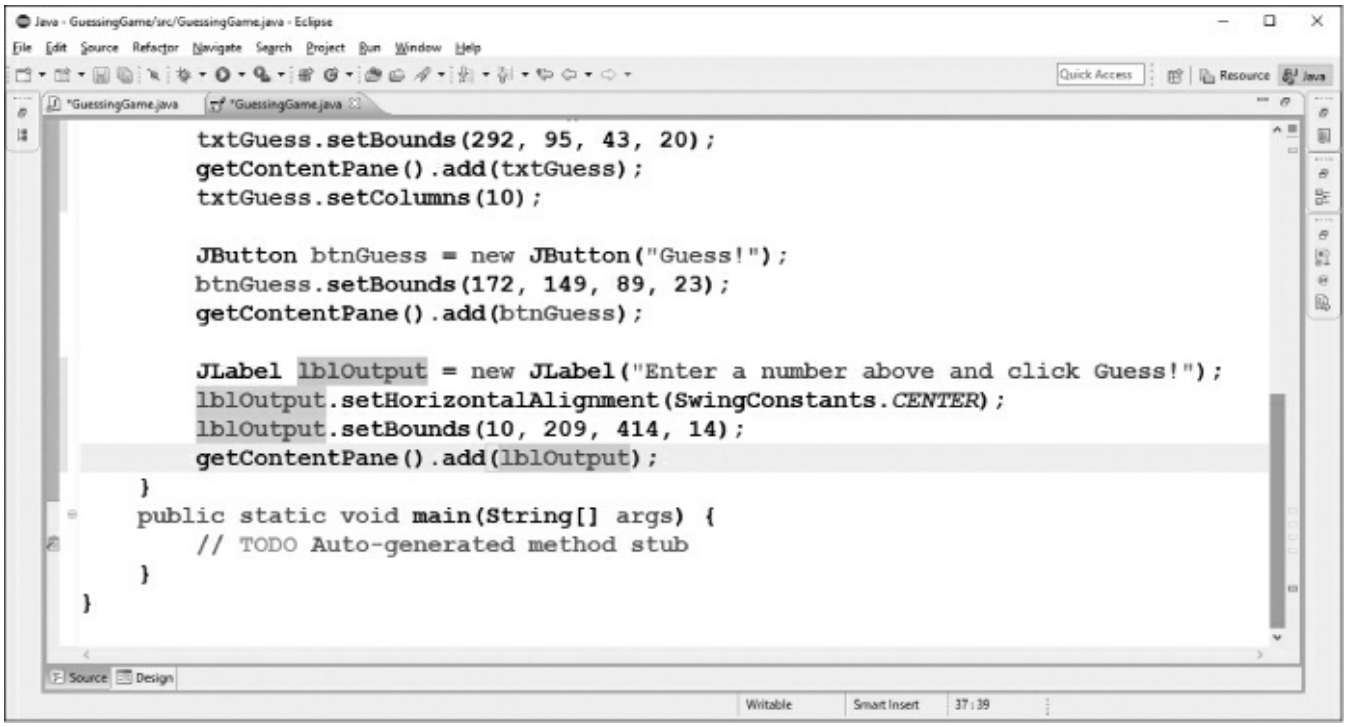


Figure 3-16: Click the Source tab, and you'll see Eclipse has coded the GUI components in Java for us.

Notice that the new variable names you gave the text field (`txtGuess`) and output label (`lblOutput`) are both visible in the source code. Now is a good time to save the file with all the changes you've made so far.

But before we start writing the code for the guessing game app, we're going to make an adjustment to the source code.

Connecting the GUI to Your Java Code

Scroll up to the top of the `GuessingGame` class, and you'll see that Eclipse has declared the `JTextField txtGuess` right at the top of the `GuessingGame` class:

```
public class GuessingGame extends JFrame {
    private JTextField txtGuess;
```

Eclipse does this for text fields by default for a reason. Declaring text fields at the top level of the class makes it possible to access the text field (to see if the user's guess is too high or too low, for example) from *anywhere* inside the class. Text fields are usually used in this way, so Eclipse declares a `private JTextField txtGuess` right at the top of the `GuessingGame` class instead of in a method or function further down in the class. The modifier `private` is used as an object-oriented programming *best practice* (a recommended way of doing things). Making something `private` keeps other classes from being able to see that part of the code. Since `txtGuess` is `private`, classes other than `GuessingGame` won't be able to access it or accidentally change the `txtGuess` variable, which is what we want.

In this guessing game app, we want to be able to access the value the user enters into the `txtGuess` text field, so it's great that Eclipse has declared it at the top level of the class. We also want to access the label `lblOutput` to be able to change the text displayed to the user (when their guess is too high, too low, or correct). So we need to add one more declaration to the top of the class, right under the declaration of `txtGuess`:

```
public class GuessingGame extends JFrame {
    private JTextField txtGuess;
    private JLabel lblOutput;
```

This line of code declares `lblOutput` as a variable that refers to a `JLabel` object in the GUI layout for the app. We've chosen to make this object reference `private` to hide this data from outside programs, but we'll still be able to use it throughout the `GuessingGame` class.

One final change is to correct the line where `lblOutput` was originally declared, down near the bottom of the class. Find the line that looks like this, further down in the code:

```
  JLabel lblOutput = new JLabel("Enter a number above and click Guess!");
```

And change it to this:

```
  lblOutput = new JLabel("Enter a number above and click Guess!");
```

Notice we've removed the additional `JLabel` declaration. If you leave this second `JLabel` declaration in the program, your app won't work because you've already declared `lblOutput` to be of type `JLabel` at the top of your class. If you leave the second `JLabel` declaration, Java will think you mean to have *two* separate `JLabel` objects named `lblOutput`. By removing the second `JLabel` declaration, you're telling Java to use the *one* `JLabel lblOutput` you've created at the top of the program and initialize it with a text value of `"Enter a number above and click Guess!"`.

While we're declaring variables at the top level of the `GuessingGame` class, let's add one more important variable: `theNumber`. Remember in the text-based version of the game, this is what we named the secret, random number the user is trying to guess. Add a declaration for `theNumber` after `txtGuess` and `lblOutput` as follows:

```
public class GuessingGame extends JFrame {
    private JTextField txtGuess;
    private JLabel lblOutput;
    private int theNumber;
```

Now that our variables are correctly declared at the top of the class, we're ready to begin coding the GUI version of the guessing game app. In the next section, you'll learn how we can get the user's guess from the GUI text field and check to see if it's too high or too low. We'll also show the output text in the GUI label `lblOutput` to help the user make their next guess.

Adding a Method to Check the Player's Guess

Now that the GUI is wired to the Java source code with the private variables we added, `txtGuess` and `lblOutput`, we'll begin coding the logic of the guessing game.

Let's write a method called `checkGuess()` near the top of the `GuessingGame` class. The signature, or skeleton, of this method will look like this:

```
public class GuessingGame extends JFrame {
    private JTextField txtGuess;
    private JLabel lblOutput;
    private int theNumber;
    public void checkGuess() {
    }
```

Notice that the method is declared as `public`. Variables in a class are usually declared as `private`, but methods or functions that operate on those variables are

usually declared as `public` so other classes can call on them. You can think of it like your checking account. Your balance is private and can only be accessed by you or the bank, but the function of making deposits is public (other people can deposit money into your account).

The second term, `void`, tells Java that we're not expecting a *return* value, which is a value that a function or method outputs, from this function. For example, a conversion calculator for Fahrenheit to Celsius called `convertFtoC()` could take a value representing a temperature in Fahrenheit and return a numeric value representing the converted temperature in Celsius. In this guessing game application, the `checkGuess()` method will not return information like this to the rest of the program. Instead, we're going to program the app to display information to the user through the graphical user interface, so we use the `void` return type, which doesn't return a value.

Finally, we've named the method `checkGuess()` and provided the opening and closing braces that will contain the code that tells the program how we want to check a user's guess.

Getting Text from a JTextField

Let's start building the `checkGuess()` method, which will take the string the user enters and check it against the number to guess. Add this line between the two braces for the `checkGuess()` method:

```
public void checkGuess() {
    String guessText = txtGuess.getText();
}
```

This line creates a new `String` variable called `guessText` that will hold the number the user enters into the GUI text field. To get the text the user entered, we'll need to use the `getText()` method by calling `txtGuess.getText()` and storing the result in the new string variable called `guessText`.

You may see a window pop up near where you're typing when you add the dot operator (.) after `txtGuess`, as shown in Figure 3-17.

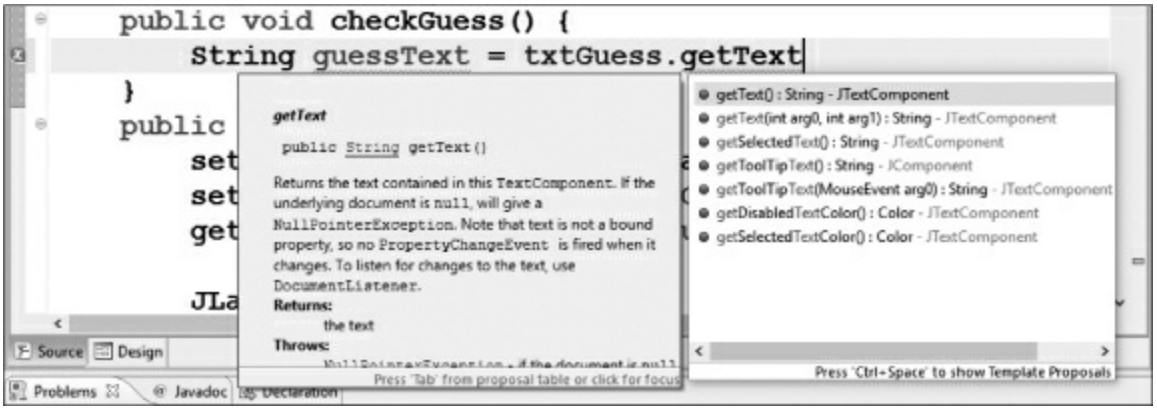

public void checkGuess() {
String guessText = txtGuess.getText
}
public
set
set
get
JLa
getText
public String getText()
Returns the text contained in this TextComponent. If the underlying document is null, will give a NullPointerException. Note that text is not a bound property, so no PropertyChangeEvent is fired when it changes. To listen for changes to the text, use DocumentListener.
Returns:
the text
Throws:
getText() : String - JTextComponent
getText(int arg0, int arg1) : String - JTextComponent
getSelectedText() : String - JTextComponent
getToolTipText() : String - JComponent
getToolTipText(MouseEvent arg0) : String - JTextComponent
getDisabledTextColor() : Color - JTextComponent
getSelectedTextColor() : Color - JTextComponent
Press 'Ctrl+Space' to show Template Proposals
Source
Design
Problems
Javadoc
Declaration

Figure 3-17: After you type txtGuess and the dot operator (.), you may see a helpful code recommendation window appear.

This is known as a *content assist*—Eclipse is trying to help by suggesting lines of code to finish the statement you're working on. This particular kind of content assist is also called a *code recommender*, because it recommends common code options, like methods in the current class or object. Code recommendation is one feature that makes Eclipse a powerful IDE for professional Java developers. It's not only faster in most cases to select a code recommender suggestion, but doing so also helps prevent errors and makes coders more efficient as they develop complex applications.

Now, we need to create a string to hold the message telling the user whether their guess was too high, too low, or correct. Let's declare one additional string:

```
public void checkGuess() {
    String guessText = txtGuess.getText();
    String message = "";
}
```

We don't know yet whether the user's guess is too high or too low, so we've initialized `message` to an empty string with a pair of double quotes (`""`). After we test the user's guess against the random number, or `theNumber`, we'll update this string with more information before we output it in the GUI label. To test the user's guess, we'll need to turn their text entry, currently stored in `guessText`, into numeric form so that we can compare it to `theNumber`.

Converting Strings to Numbers

In the text-based version of this app we made in Chapter 2, we used a `Scanner` object to scan a number value from the keyboard text the user entered. A scanner could work in a GUI, but there's a more compact way to do this that will work

better for this version.

Just like the `Math` class had `Math.random()` so that we could generate a random number, there is also a class for working with integers, called `Integer`. The `Integer` class has several functions that are useful for working with whole numbers, including a method to look for integers in strings of text, `Integer.parseInt()`. Enter the following line below `String message = ""`.

```
public void checkGuess() {
    String guessText = txtGuess.getText();
    String message = "";
    int guess = Integer.parseInt(guessText);
}
```

First, this line declares `guess` to be an integer variable. Then, it searches for, or *parses*, an integer from the text the user entered. For example, the string `"50"` will become the *number* `50`. Finally, it stores the number in the variable `guess`. We need the numeric version of the user's guess in order to compare it to `theNumber`, using the comparison operators `<` and `>`.

With the user's guess stored in the variable `guess`, we're ready to test it against the computer's secret, random number `theNumber`. We haven't actually programmed `theNumber` to hold a value yet, but we'll do that soon. Comparing the guess with `theNumber` will look like it did in the text-based version of the game, except we won't print to the console using `System.out.println()`. Instead, we'll store output for the user in the string variable `message` that we created:

```
    String guessText = txtGuess.getText();
    String message = "";
    int guess = Integer.parseInt(guessText);
    if (guess < theNumber)
        message = guess + " is too low. Try again.";
    else if (guess > theNumber)
        message = guess + " is too high. Try again.";
    else
        message = guess + " is correct. You win!";
}
```

Notice the three `if-else` statements are virtually identical to those in the text-based version of the game, except that we're storing the output `message` (too high, too low, or correct) in a variable instead of outputting it directly to the console window. We still need to show `message` to the user, so we'll do that using the `lblOutput` GUI label. We'll use the `setText()` method like so:

```
    else
        message = guess + " is correct. You win!";
    lblOutput.setText(message);
}
```

This command will show the output message to the user in the GUI window by changing the text property of the `JLabel` called `lblOutput` to the correct `String` variable `message` based on the user's guess.

Your completed `checkGuess()` code should look like the following:

```
public void checkGuess() {
    String guessText = txtGuess.getText();
    String message = "";
    int guess = Integer.parseInt(guessText);
    if (guess < theNumber)
        message = guess + " is too low. Try again.";
    else if (guess > theNumber)
        message = guess + " is too high. Try again.";
    else
        message = guess + " is correct. You win!";
    lblOutput.setText(message);
}
```

Next, we need to write the code that assigns a random value to `theNumber`. We'll use the `newGame()` method, because we want to have a new random number every time the user begins a new game.

Starting a New Game

We want to tell the computer to pick a new, secret, random number whenever a new round of the guessing game starts, so it makes sense to do this in a method in the `GuessingGame` class. By doing it this way, we'll be able to call on the method every time the user wins and wants to play a new game.

The method signature of the `newGame()` method looks similar to the `checkGuess()` method. Place it just after the closing brace for `checkGuess()` and before `public GuessingGame()`:

```
    lblOutput.setText(message);
}
public void newGame() {
}
public GuessingGame() {
```

We'll make this method `public` so that it can be called from an outside class. We also need to let Java know that just like the `checkGuess()` method, this method has no information that needs to be returned, so its return type is `void`. Finally, the method's name is `newGame`, and it includes open and close parentheses, `()`. The method body is empty right now, with nothing between the braces, but we'll address that next.

A new game in this app just means that we're telling the computer to pick a new random number. The code to assign this random value to `theNumber` will look and function like it did in the text-based version of the game, except now we're putting this code inside its own, separate method. So, the finished `newGame()` method should look like this:

```
public void newGame() {
    theNumber = (int)(Math.random() * 100 + 1);
}
```

That's it! Now we have all the necessary pieces in place to build a functional guessing game app, so we just have to connect them to the GUI interface. Once the app can listen for user events so it can respond to the user clicking the "Guess!" button, we'll be ready to run the game!

Listening for User Events: Click to Guess!

The last step we need to complete before running the app is to connect the `btnGuess` button (labeled "Guess!") with the function that checks the user's guess, `checkGuess()`. With Java, we can use *event listeners* to do this. Event listeners are just code that tells the program to wait, or *listen*, for user interaction *events*, like button clicks, mouse movement, or keyboard entries.

Go back to the GUI preview of the app by clicking on the Design tab. In the GUI preview window, find the "Guess!" button, as shown in Figure 3-18, and double-click it.

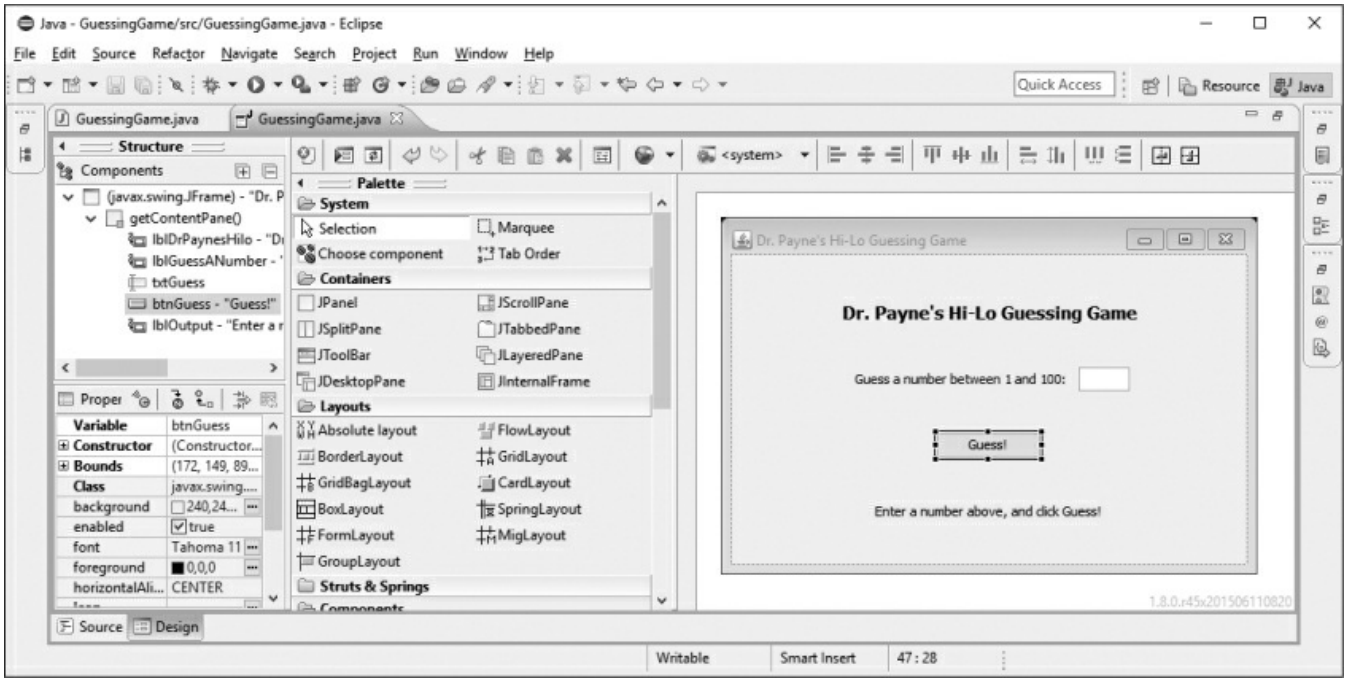


Figure 3-18: In the Design view, double-click the "Guess!" button to add an event listener to the code.

Double-clicking the "Guess!" button automatically opens the Source tab, where you'll notice Eclipse has added new code. This new code is the skeleton of an event listener—in this case, an `ActionListener()` for the event that occurs when the user takes an action on the `JButton btnGuess`. In this case, the action is clicking the button.

Eclipse has added everything except the action we want to perform when the user clicks the button. Let's take a look at the event listener code Eclipse has created for us.

```
    JButton btnGuess = new JButton("Guess!");
❶ btnGuess.addActionListener(new ❷ActionListener() {
    ❸ public void actionPerformed(ActionEvent e) {
      }
});
btnGuess.setBounds(172, 149, 89, 23);
getContentPane().add(btnGuess);
```

The event listener for an `ActionListener()`, the specific listener that handles button clicks, is connected to `btnGuess` with the `addActionListener()` method at ❶. The next couple of lines of code are all built inside the parentheses for the `addActionListener()` function.

Inside the `addActionListener()` method, the first thing we see is the keyword `new`, letting us know that Eclipse is creating a new object of the `ActionListener`

class at ❷. What makes this new object different from the ones we've seen in the past, though, is that it's an *anonymous inner class*, meaning that it doesn't have a variable that refers to it and it's created entirely *inside* the `GuessingGame` class. Anonymous inner classes can help us code faster since we don't need to build a completely separate class to handle short, quick events like a button click. Instead, we can insert the code for an event handler right where we create the button. In this case, the `ActionListener` will be listening for just a button click, so it's a perfect candidate for an anonymous inner class.

Eclipse creates the skeleton for the anonymous inner class, but we still need to code the button's behavior. This next section of code will go inside the braces of the `actionPerformed()` method at ❸. The `actionPerformed()` method is where we describe what we want the app to do when the user has clicked `btnGuess`. The method accepts an `ActionEvent` as an argument, which is the event, or user action, the listener is listening for. In this case we've named the `ActionEvent e`, but you can use any variable name you'd like. When the user enters a guess and clicks the "Guess!" button, that action is assigned to `e`, and we need to check their guess with the `checkGuess()` method we created earlier:

```
    btnGuess.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            checkGuess();
        }
    });
```

Thanks to Eclipse, the only part we had to add was the `checkGuess();` command. The automatic code completion feature has saved us time and helped us avoid typographical errors.

With this section of code in place, now we just need to tell the app what to do on startup. Take a moment to save your work so far.

Setting Up the GUI Window

Before we run the game for the first time, we need to tell the program what to do to set up the GUI window. This includes building the GUI interface, starting a new game by calling the `newGame()` method, and setting the desired width and height dimensions of the window.

Inside the braces for the `main()` method near the bottom of the file, add the

following lines of code:

```
    public static void main(String[] args) {
     ❶ GuessingGame theGame = new GuessingGame();
     ❷ theGame.newGame();
    }
}
```

The line of code at ❶ creates a new `GuessingGame` called `theGame`. The line at ❷ will start the guessing game with a new random number.

Now, let's give Java an idea of the size we want the window to be. For a GUI desktop app, we can specify the size of the frame for the app using the `setSize()` method, entering the width and height of the window we want to create.

```
    public static void main(String[] args) {
        GuessingGame theGame = new GuessingGame();
        theGame.newGame();
        theGame.setSize(new Dimension(450,300));
```

The `setSize()` method requires a parameter of the `Dimension` type that's built into the `java.awt` library (`.awt` is short for *abstract window toolkit*). The `new Dimension()` inside the parentheses for `setSize()` tells Java to create an object of type `Dimension` with width and height parameters of 450 pixels wide by 300 pixels tall. But we haven't imported the `java.awt` library yet, which means Eclipse doesn't know what `Dimension` refers to, so your Eclipse editor will tell you there's an error with this statement by underlining the class name `Dimension` in red. If you click the word `Dimension` in the editor, Eclipse shows you a list of Quick Fix options, as shown in Figure 3-19. (If you aren't able to get the Quick Fix pop-up to display, press CTRL-1 or ⌘-1 after placing your cursor in `Dimension`.)

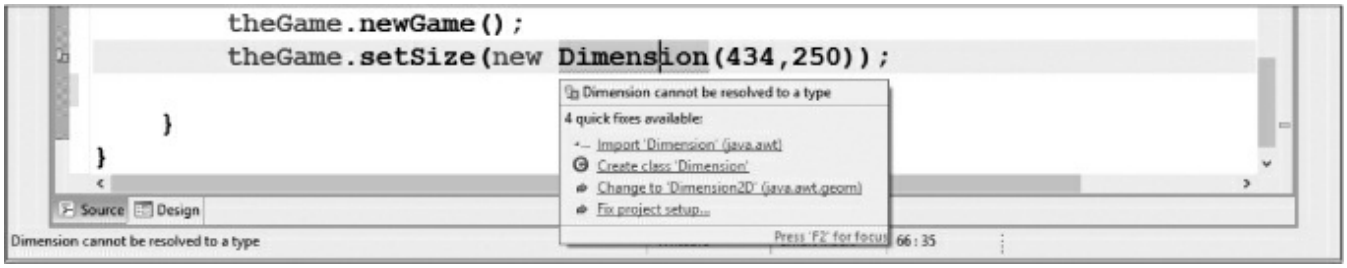


Figure 3-19: Eclipse recommends Quick Fix options for the new code we've entered; choose the top option to import the correct class.

We want to make use of the `Dimension` class that's already provided in the `java.awt` package, so the topmost Quick Fix option, **Import 'Dimension'**

(java.awt), is the right one for this situation. Click it, and Eclipse will add `java.awt.Dimension` to the list of imported classes at the top of the file.

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import java.awt.Dimension;
import java.awt.Font;
import javax.swing.JTextField;
```

Finally, add this line to the `main()` method to make the app visible on the screen:

```
        theGame.setSize(new Dimension(450,300));
        theGame.setVisible(true);
    }
```

This command calls the `setVisible()` method on the `GuessingGame` object `theGame`. Remember, `GuessingGame` extended the class `JFrame`, so `theGame` is also a *descendent* of the `JFrame` parent class. We want to set the visible property of the main `JFrame` holding the GUI version of the guessing game to `true`, which is a Boolean value.

There are only two *Boolean values*, `true` and `false` (both spelled with all lowercase letters), and these are used in Boolean expressions. *Boolean expressions* are any set of arguments that can be evaluated to `true` or `false`. The `if` statements we used in the `checkGuess()` method, like the expression `(guess < theNumber)`, are just one example we've already seen. When we evaluate this expression and check whether `guess` is less than `theNumber`, the expression evaluates to either `true` (meaning `guess` is less than `theNumber`) or `false` (`guess` is not less than `theNumber`). The result of that Boolean expression determines whether the statement following the `if` statement will run.

The complete `main()` method for the `GuessingGame` class is as follows:

```
    public static void main(String[] args) {
        GuessingGame theGame = new GuessingGame();
        theGame.newGame();
        theGame.setSize(new Dimension(450,300));
        theGame.setVisible(true);
    }
```

Time to Play!

With your code complete, now is a good time to save the *GuessingGame.java* file. Once you've done so, click the run button or select **Run ▸ Run** from the menu. The game window will appear.

Try entering some guesses and see if the app works. While you're playing the game, check each component to make sure it works. Make sure to test the "Guess!" button, the label at the bottom, and the text field. If you spot problems, go back and check your code. Pay attention to when or where the error occurs to narrow down the source. You'll get better at finding errors as you learn to code. Figure 3-20 shows how the app might look when it's working properly.

Figure 3-20: The "Guess!" button works!

When your game is playable all the way through, you're ready to take the next step.

Right now, the game is playable, but only until you win. The current version of the app doesn't have the code to tell Java to start a new game every time we win the current one.

First, let's set it up so we can keep playing the game even after we win once, and then we'll make a few tweaks to the user interface to provide a better user experience.

Adding a Play Again Feature

Just like we did in the first text-based version of the game, let's add a play again feature. We created a `newGame()` method already, but we only called that method

when setting up the game for its initial display in the `main()` method of the app.

We need to either change the GUI or change the app behavior. We could add a Play Again button for the user to click when they want to play another round, or we could just make the app automatically start a new game every time the user wins.

The benefit of adding a Play Again button is that it would be easy to understand from the user's perspective. However, we wouldn't want the Play Again button to be visible in the GUI window all the time, because it's needed only when the user wins. One option would be to add a button but make it invisible while the game is running, having it become visible only when the user wins. In order to keep our game simple, we won't do this here, but if you're interested in trying it out, see Programming Challenge #2 on page 73.

Instead of adding a button, let's change the app's behavior so that a new number is picked automatically after the user wins the current round. This requires changing only one section of the program, the `else` statement that handles a guess matching the random number. If the user has won a round, a new round should begin automatically. This makes intuitive sense—no extra buttons, no decisions to make. If the player decides they're done playing, they can simply close the window. We'll add this behavior by changing the following lines:

```
    else
        message = guess + " is correct. You win!";
```

We'll call `newGame()` to start a new game. To add the call to the `newGame()` method inside this `else` statement, we'll group two statements (setting the variable `message` and calling `newGame()`) together as one block like so:

```
    else {
        message = guess +
                " is correct. You win! Let's play again!";
        newGame();
    }
```

The braces group these two statements as a single block of code to be executed together whenever the `else` statement is selected as the correct program path. Without the braces, only the first line after the `else` statement would be executed, and the `newGame()` method would run after the `if-else` statements are evaluated. This would make the game change the secret number every time the

user tried to guess.

Notice also that we added `Let's play again!` to the end of the message string. This lets the user know that they're able to play again just by guessing and can expect the computer to pick a new number each round. Make these changes, save the program, and run it again to see the new version of the game!

Improving the UX

As you play the game, you may notice a few elements of gameplay that could make the user interface even better. For example, if we could have the app accept a new guess either when the user clicks the "Guess!" button or when they press ENTER from the keyboard, that would provide a more natural and intuitive *user experience (UX)*.

One other problem is that when we want to guess a new number, we have to highlight and select the old number (or, alternatively, press the DELETE or BACKSPACE key) to clear the previous guess. We also have to click back into the text field, because the focus switches to the "Guess!" button. To improve the user interface, we could have the cursor return to the text field after each guess and automatically highlight or delete the previous guess's text so that the new guess replaces the old one.

Let's tackle these two items now.

Allowing Users to Press Enter to Guess

The first improvement we'll make is to allow the user to press ENTER

to submit their guess. First, go to the Design tab and right-click the "Guess!" button. In the pop-up menu, select **Add event handler ▸ action ▸ actionPerformed**.

Just like when we created an event handler for `btnGuess`, Eclipse will automatically add the code to create an action listener for `txtGuess`, as shown in the following code snippet. And, just like we did before, we'll add the `checkGuess()` method inside the braces of the `actionPerformed()` portion of this code. The only action event we usually want to handle from inside a text field is the user pressing the ENTER key.

```
    txtGuess.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
            checkGuess();
        }
    });
```

After you add this line, save the file and run it again. Now when we press the ENTER key, it activates the `checkGuess()` method. The "Guess!" button still works, so the user has their choice of interacting only by keyboard or with their mouse.

Automatically Removing Old Guesses

Now let's solve the issue of having to click back into the text field to delete our old guesses. To address this, we'll need to learn a couple of new methods for working with `JTextField` objects like `txtGuess`.

First we have to consider when we want text to be selected. For the guessing game, after the user's guess is evaluated is probably the right time to enable the user to guess again. In the code, this would be at the end of the `checkGuess()` method, as we want it to occur after the user has made one guess and is ready to make another.

So, we know we need to add code inside the end of the `checkGuess()` method to place the cursor back into the `txtGuess` text field and select the text from the user's previous guess. But how do we know which commands would place the cursor in the text field and select all the text? This is a great time to use the content assist feature in Eclipse. Type the name of the object, **`txtGuess`**, followed by the dot operator (.), and Eclipse will display a list of available methods, as shown in Figure 3-21.

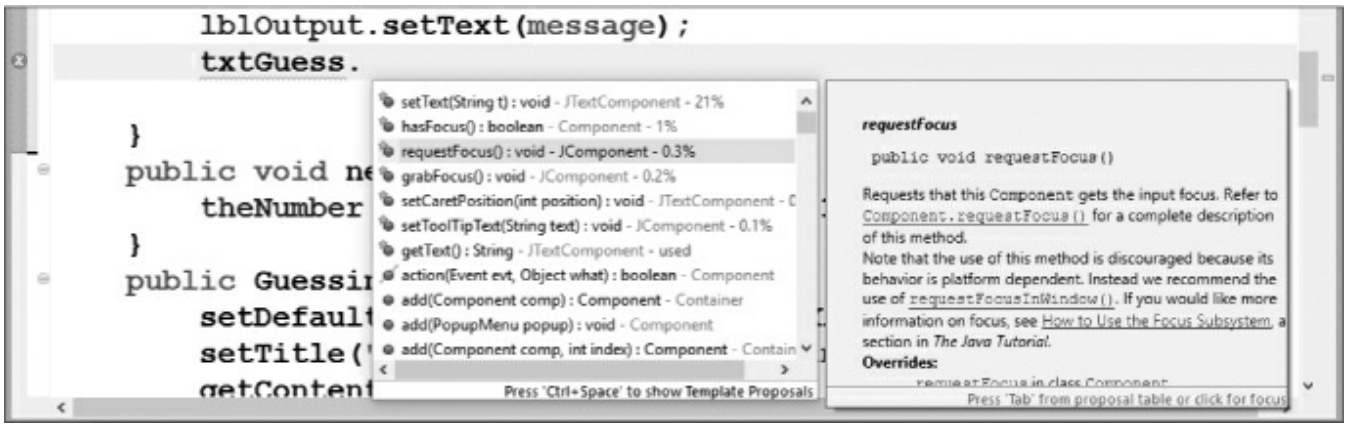

lblOutput.setText(message);
txtGuess.
setText(String t) : void - JTextComponent - 21%
hasFocus() : boolean - Component - 1%
requestFocus() : void - JComponent - 0.3%
grabFocus() : void - JComponent - 0.2%
setCaretPosition(int position) : void - JTextComponent
setToolTipText(String text) : void - JComponent - 0.1%
getText() : String - JTextComponent - used
action(Event evt, Object what) : boolean - Component
add(Component comp) : Component - Container
add(PopupMenu popup) : void - Component
add(Component comp, int index) : Component - Contain
Press 'Ctrl+Space' to show Template Proposals
requestFocus
public void requestFocus()
Requests that this Component gets the input focus. Refer to Component.requestFocus() for a complete description of this method.
Note that the use of this method is discouraged because its behavior is platform dependent. Instead we recommend the use of requestFocusInWindow(). If you would like more information on focus, see How to Use the Focus Subsystem, a section in The Java Tutorial.
Overrides:
Press 'Tab' from proposal table or click for focus

Figure 3-21: The content assist feature displays all the methods available for the JTextField txtGuess.

You'll see that there are dozens of methods available for working with `JTextFields` like `txtGuess`, but the one we're looking for is `requestFocus()`. The documentation for this function says, "Requests that this Component gets the input focus." The *focus* in this case is the cursor. This method will do exactly what we want and request that the cursor be placed back in the text field every time the user guesses again.

Finally, we want to select all the text from the user's previous guess so that their new guess will replace the old one automatically. To select all the text in a text field, use the `selectAll()` method, which is also listed in Eclipse's content assist feature suggestions. Add both these lines to the end of the `checkGuess()` method, right after the line for `lblOutput.setText(message);` and right before the closing brace:

```
        lblOutput.setText(message);
        txtGuess.requestFocus();
        txtGuess.selectAll();
    }
```

Now is a good time to save your application. When you try running the game again, you should feel a much smoother, more intuitive user experience. The guessing game finally has an elegant, easy-to-use GUI that provides a simple UX. Our game is easy to play, but a challenge to master. It's just like the games that we play for fun, but we've built this one from scratch, using Java and the Swing toolkit for a professional appearance.

Now, we have a great app all around, except for one important thing: security. You may ask, "What? Security? I thought we were just building a fun GUI game!" And that's true—the program works beautifully, but only as long as the user enters their numbers correctly.

What happens if the user's fingers slip and they press ENTER or click the "Guess!" button before they've typed their guess? What if they enter any noninteger text, like **`blah`**, as a guess? Figure 3-22 shows what happens in Eclipse when the user enters bad input.

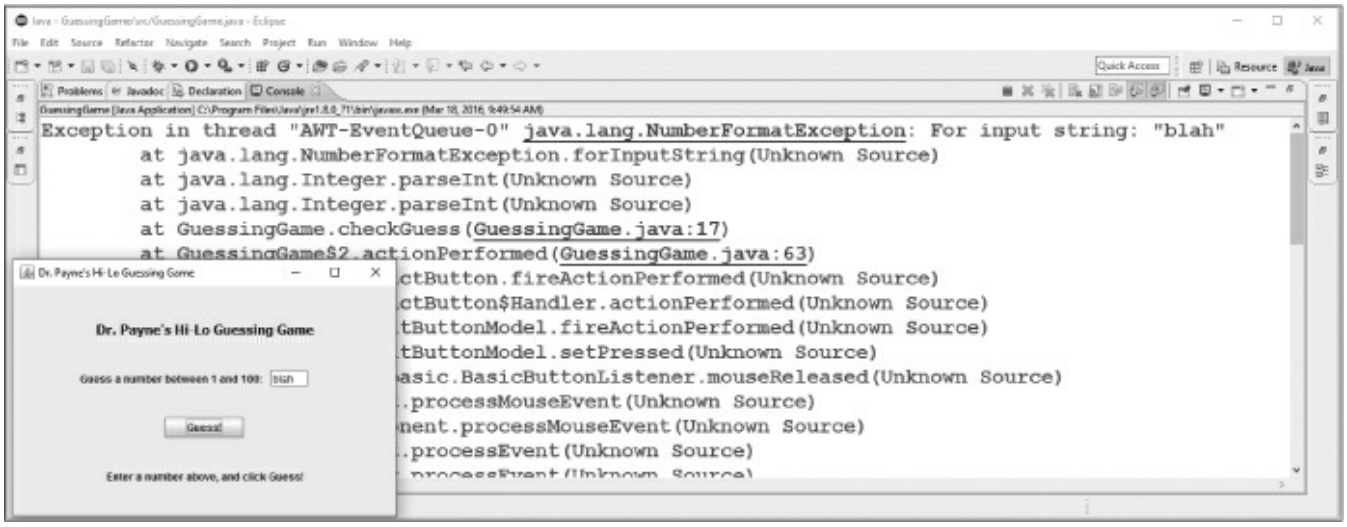

Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException: For input string: "blah"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at GuessingGame.checkGuess(GuessingGame.java:17)
Dr. Payne's Hi-Lo Guessing Game
Dr. Payne's Hi-Lo Guessing Game
Guess a number between 1 and 100:
blah
Guess!
Enter a number above, and click Guess!

Figure 3-22: Bad things happen when the user enters bad data, like blah, in the text field.

As you can see, our console window is now filled with errors. The bad input resulted in an *exception*—an unexpected or faulty programming situation. The first line of output in Figure 3-22 tells us there was a `NumberFormatException` because the string `"blah"` is not recognized as a number. You'll need to learn how to handle exceptions in the programs you create.

Handling Bad User Input

Java provides a safe way of anticipating and dealing with exceptions, the bad things that can happen when a program runs. In addition to bad user input like we just discussed, exceptions can include missing files (a file gets deleted, or you remove a USB drive unexpectedly) or network problems (such as a missing connection or a bad server).

We call the code built for dealing with exceptions an *exception handler*. An exception handler deals with exceptional situations. But how do we know when an exception might occur? Do we have to predict everything bad that could happen when a program is running?

Many of the situations in which errors could occur are particular types of events, like user input, file input/output, or network access. For example, think about the exceptions we listed previously. Bad user input could happen any time we ask the user for input of a certain type. A missing file could be a problem whenever we're reading or writing files; for example, when viewing and saving pictures from a photo app. If we can anticipate possible exceptions and write

exception handlers to deal with those exceptions, our code will be safer, more secure from malicious hackers who might want to crash your program by entering bad input, and more reliable.

In the case of the guessing game, the exception occurred when we asked the user for a number, but they entered text instead. Specifically, the exception was triggered when we tried to parse an integer out of a string that didn't contain an integer. User input is a common source of exceptions, both accidental and intentional. An example of the latter is a hacker trying to break your program.

Java gives us a clean way of handling exceptions with the `try-catch` statement. The idea of a `try-catch` block is that we want the program to *try* to do something that we know might cause an exception and *catch*, or handle, the exception that might occur rather than allowing it to break the program. The `catch` block executes only when an exception happens.

The last part of a `try-catch` is the optional `finally` block. In a `try-catch-finally` statement, the `finally` block will *always* execute if it's provided, regardless of whether there's an exception. If there's no exception, the `finally` block will still run after the `try` block finishes. If an exception occurs, the `finally` block will run after the `catch` block has completed.

The format of a `try-catch` typically looks something like this (don't type this into your code; this is just an example of the format):

```
try {
    // Something that could cause an exception, like getting bad input
} catch (Exception e) {
    // Handle or recover from the exception, like asking for good input
} finally {
    // Any wrap-up code; this will always run last, exception or not
}
```

Notice we group statements for each part of the `try-catch-finally` statement with braces, just like we do for `if-else` and loop statements. The `catch` part requires an `Exception` parameter; this can be a specific exception, like `NumberFormatException`, or we can use the base class `Exception`. This parameter will receive information about the `Exception`, like what line number the exception occurred on.

To place the `try-catch-finally` blocks correctly in our game's code, we need to think about where the exception might occur (and wrap that code inside a `try` block), what we want to do if there is an exception (and put that in a `catch` block),

and what we want to do after either the `try` or `catch` block finishes (this part goes inside the `finally` block).

Because the user input problem occurred when we tried to parse an integer out of text the user entered, we want to wrap that code in a `try` block. Let's put the `try` keyword and an opening brace right before the line that parses the integer out of the text:

```
    public void checkGuess() {
        String guessText = txtGuess.getText();
        String message = "";
        try {
            int guess = Integer.parseInt(guessText);
            if (guess < theNumber)
```

If the user didn't enter a number, we can't test for too high or too low, so the `try` clause will go all the way to the end of this pair of `if-else` statements. Place a closing brace for the `try` block after the closing brace for the `else` statement, as shown in the following listing. The `catch` keyword should go next. In this game, if a user enters bad input, we just want to ask them to enter a whole number between `1` and `100`. We can do that with the `message` string that gets displayed in `lblOutput` later. So, the `catch` block should look like this:

```
        else {
            message = guess +
                    " is correct. You win! Let's play again!";
            newGame();
        }
    } catch (Exception e) {
        message = "Enter a whole number between 1 and 100.";
    }
```

The last step we want to take, whether there was an exception or not, is to show the output message, request the focus for the text field, and select all the text to prepare for the next entry. These three statements will go into the `finally` block, after the closing brace of the `catch` statement we just added:

```
      } finally {
          lblOutput.setText(message);
          txtGuess.requestFocus();
          txtGuess.selectAll();
    }
}
public void newGame() {
```

You may notice that the statements inside each of the `try-catch-finally` blocks are indented to make the code easier to read. Remember, there's a shortcut in Eclipse that will automatically indent everything correctly after you've made any changes. Just select all the code, then go to **Source ▸ Correct Indentation**.

Save your app and run it a few times to test the user interface. Test a few nonintegers to make sure the exception handling works the way it's supposed to. If you run into errors that prevent the file from compiling, check your braces. Whenever we add blocks around existing code, braces are the first place to check for errors. Once your `try-catch-finally` statements are in place, you've got a more reliable, more predictable, and safer version of the GUI guessing game app.

Safe programming isn't just a nice touch—it's often a critical part of writing good software. For our guessing game, the consequences of bad code don't seem dire. But think about the code running on a medical device, a satellite, a car, an airplane, or the power grid that delivers electricity to your home and city. Secure code is vital to our safety and security. And even a small game or app running unsecure code on your smartphone could expose your data to hackers. Safe programming is always a smart practice.

What You Learned

In the GUI version of the guessing game app, we've grown our Java skills to include all the following:

- Creating GUI windows by importing the `javax.swing` package
- Setting up a GUI app in Eclipse
- Designing a GUI with the WindowBuilder Editor
- Using the Properties pane to change GUI component properties
- Naming GUI components consistently for easy use in Java code
- Connecting GUI elements to Java variables and programming code
- Getting user text from a `JTextField` and using it in a program
- Listening for user events, like button clicks
- Improving the user experience, or UX, by testing the app with the user in

mind

- Handling exceptions with `try-catch-finally`

Programming Challenges

Try these challenges to review and practice what you've learned and to expand your programming skills by trying something new. If you get stuck, visit *https://www.nostarch.com/learnjava/* for sample solutions.

#1: Showing Users How Many Tries They Took

Try changing the winning message to tell the user how many tries they took when they win, as in the following:

```
62 is correct! You win after 7 tries!
```

To accomplish this task, you'll need to create a new variable (like `numberOfTries`) to store how many times the user guessed, add to the number of tries every time the `checkGuess()` method runs successfully, and display the number of tries to the user when they win.

As an alternative version, you could begin the game with seven or eight tries and then subtract a try every time the user guesses. When the last guess has been used, you could tell the user they lost, show them what the correct number was, and start a new game. Give it a shot!

#2: Showing and Hiding a Play Again Button

When we built the GUI for this chapter's guessing game, we decided not to include a Play Again button because it would have cluttered the interface when it wasn't in use and it would really only be needed at the end of the game. As it turns out, we can hide a GUI component with the `setVisible(false)` method. Then we can show it again with `setVisible(true)`, just as we did with our `JFrame`.

First, we could add a Play Again button, called `btnPlayAgain`, to the GUI layout. In the code after the button is created, we could call `btnPlayAgain .setVisible(false)` to hide the button. In the winning `else` statement in `checkGuess()`, we could call `btnPlayAgain.setVisible(true)` to show the button instead of starting a new game. Once you've written that code, remember to

double-click the button in the GUI preview to add an event listener, call `newGame()` in that listener, and then hide the button again.

One important note before you start: unlike with the other button (`btnGuess`), you need to change the the `visible` property to hide and show the button from several places in the code. As with `txtGuess` and `lblOutput`, you'll need to declare `private JButton btnPlayAgain` at the top of the class, but make sure not to redeclare the button in the code below it.

When you get it working, the game should start with the Play Again button hidden, and the button should appear when you win the game. When you click the button, a new game will start, and the button will disappear again. Play the game with the button, and if you think of any other improvements for the UX, go ahead and try them!

#3: Creating a GUI MadLib

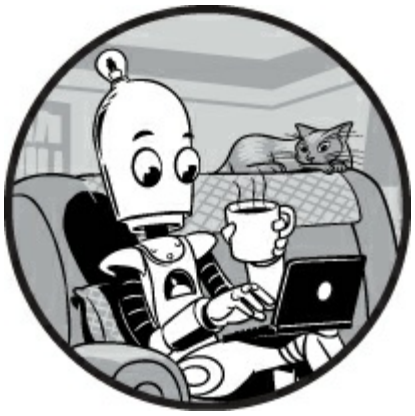
Look back at the *MadLibs.java* program you built in Chapter 2 (Programming Challenge #3 on page 41) and create a new *MadLibGUI.java* app that prompts the user for several words in a graphical user interface with labels and text fields, like `txtColor`, `txtPastTenseVerb`, `txtAdjective`, and `txtNoun`. Add a button that the user can press to generate a MadLibs-style story.

Explore and play around with several more properties of the GUI elements in the WindowBuilder Editor, including background colors, fonts, initial text values, and more. When the user clicks the button, the program should display the completed MadLibs story in a `JLabel` or a `JTextArea`, as shown in Figure 3-23.

Figure 3-23: A GUI MadLibs app

4

CREATING YOUR FIRST ANDROID APP



To build the mobile version of the Hi-Lo guessing game app, we're going to use Android Studio. If you didn't set this up in Chapter 1, go back to "Installing Android Studio for Mobile App Development" on page 8 and set it up. As with the GUI version of the game, we're going to create a user interface for the app, as shown in Figure 4-1. The apps you've created so far could only be run on your desktop computer, but Android apps can run on any Android device, including phones, tablets, watches, TVs, and so on. If you don't own an Android device, don't worry—you can still code for Android, as the Android emulator that comes with Android Studio lets you simulate running a device.

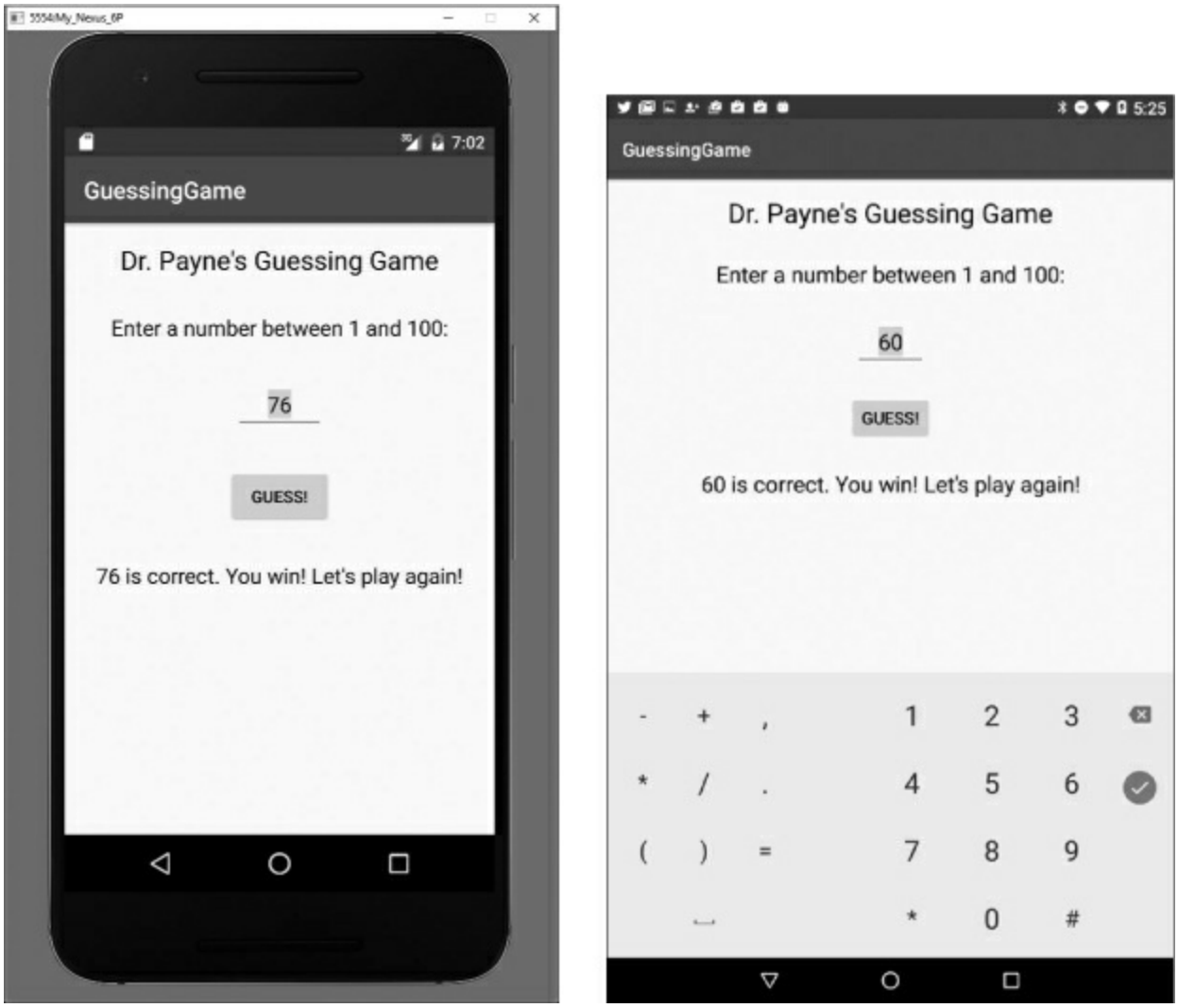


Figure 4-1: The guessing game will become a mobile app in this chapter. The app runs on a simulated Android phone (left) and on an actual Android tablet (right).

Figure 4-1 shows our app running on the Android emulator, simulating how the app looks on a Nexus 6P, and on an Android tablet. Both versions are running the same code—and, even more conveniently, we can reuse a lot of the Java code we built for the previous desktop versions because Android is built on Java!

Notice that the emulator and the actual device look very similar, with some minor differences to account for the different screen sizes between the phone and the 7-inch tablet. In the same way that you can write a Java app for Windows and run it on macOS or Linux, you can write an Android mobile app in Java and run it on any of literally thousands of Android devices. Let's get started building your first Android mobile app!

Starting a New Android Studio App Project

The first time you start Android Studio, it may take a few minutes to update and run. When it starts, you'll see a screen like the one shown in Figure 4-2. Select the option labeled **Start a new Android Studio project**.

Name your new project *GuessingGame*, with no space. If you have a website, you can enter it in the Company Domain field, as shown in Figure 4-3. Otherwise, leave the default example domain. Next, choose a project location. I've set up a folder called *AndroidProjects* to keep things organized, and you should do the same.

Welcome to Android Studio
Android Studio
Version 2.3
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Import project (Eclipse ADT, Gradle, etc.)
Import an Android code sample
Configure
Get Help

Figure 4-2: Once Android Studio finishes setting up, choose ***Start a new Android Studio project****.*

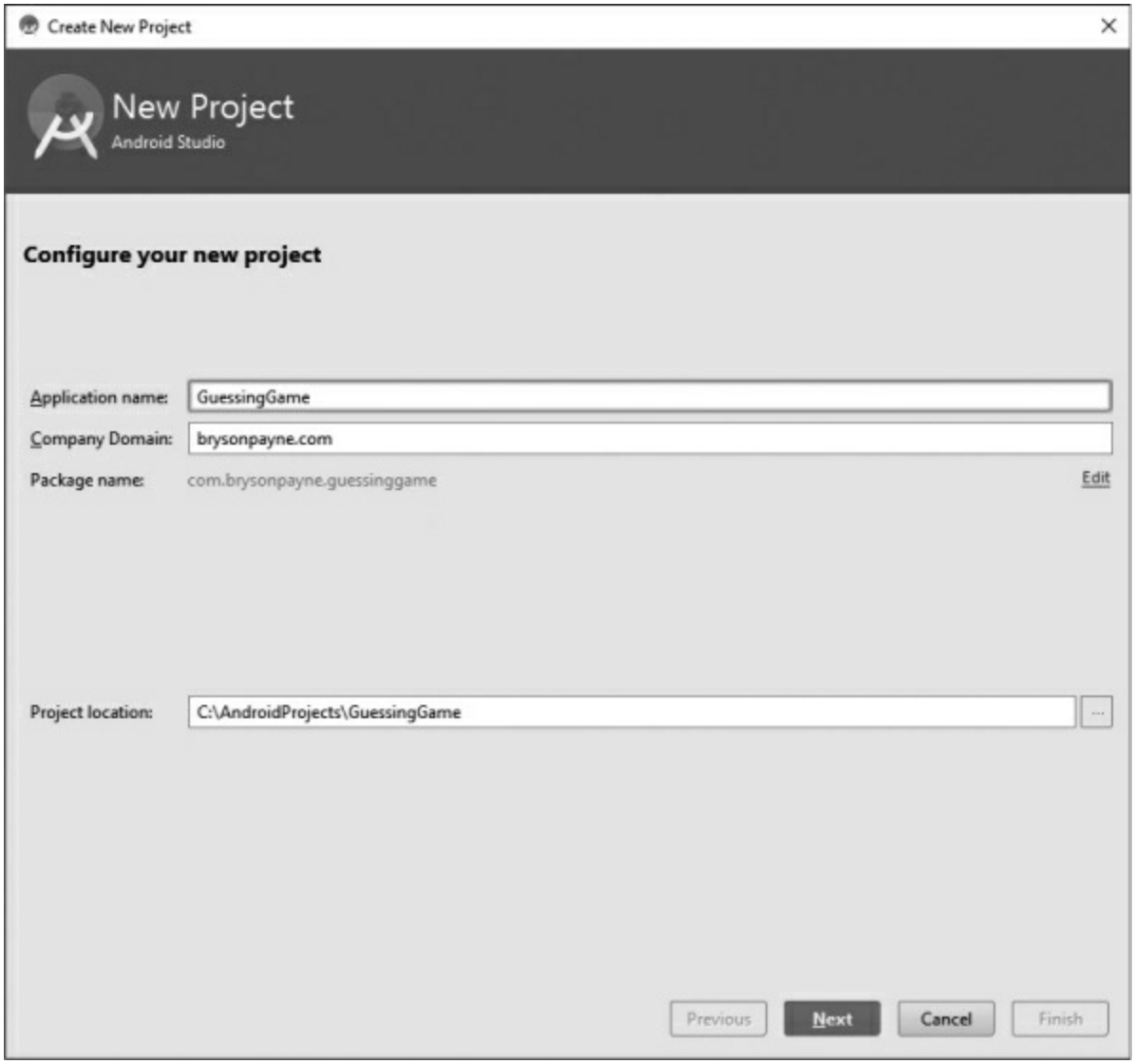


Figure 4-3: Name your new Android Studio project GuessingGame, *with no space.*

Click **Next** after naming your project. Now you'll have the opportunity to choose which Android version number your app should run on. The Android platform evolves rapidly as new devices and capabilities are developed, so we'll need to pick a target. Fortunately, Android Studio is regularly updated with both the latest Android versions and information on how many devices still use the older versions. You'll also need to select a *software development kit (SDK)* or *application program interface (API)* level when you choose which versions of Android to support. SDK and API levels include tools you'll use to develop your app, but they're tied to specific versions of Android. The Target Android Devices window, shown in Figure 4-4, allows you to choose a minimum SDK or API level for each app you develop.

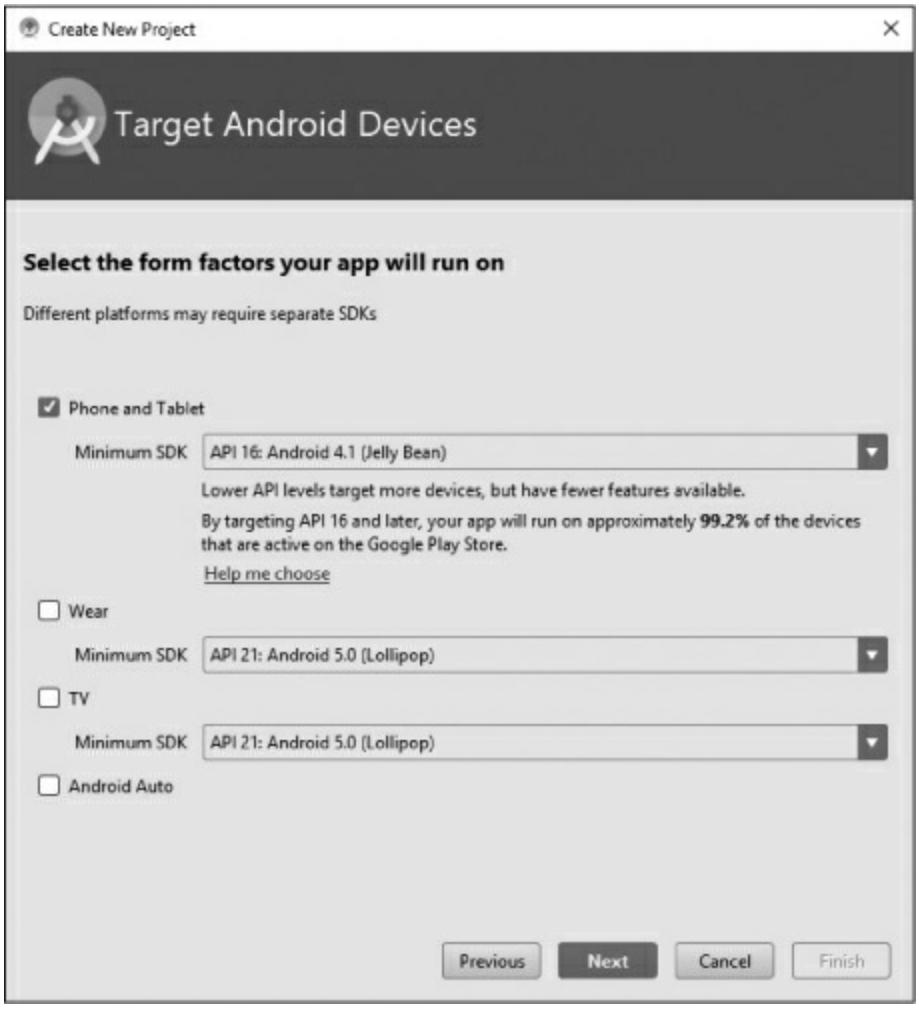

Create New Project
Target Android Devices
Select the form factors your app will run on
Different platforms may require separate SDKs
Phone and Tablet
Minimum SDK
API 16: Android 4.1 (Jelly Bean)
Lower API levels target more devices, but have fewer features available.
By targeting API 16 and later, your app will run on approximately 99.2% of the devices that are active on the Google Play Store.
Help me choose
Wear
Minimum SDK
API 21: Android 5.0 (Lollipop)
TV
Minimum SDK
API 21: Android 5.0 (Lollipop)
Android Auto
Previous
Next
Cancel
Finish

Figure 4-4: Choose the Minimum SDK level for the Android devices you're developing for.

Choose **API 16 (Android 4.1, Jelly Bean)** as the minimum SDK for the guessing game app. Android Studio will tell you that this runs on more than 99 percent of active Android devices. Click **Next**.

The New Project wizard will ask you what kind of *activity* you want to add to your app. An activity is just something the user can do. Each activity usually has its own screen layout with a user interface, like the ones shown in Figure 4-5.

There are lots of choices for the main activity for an app, such as Google Maps, Login, Settings, Tabbed Activities, and more. The Basic Activity will give us a great foundation for the guessing game app, so choose that option and click **Next**.

We'll use the default names provided by Android Studio on the following screen (shown in Figure 4-6): *MainActivity* for the activity name and *activity_main* for the layout name. The activity file contains the Java source code that runs the app. The layout is a separate file that contains the app interface.

Click **Finish** to complete your app project setup.

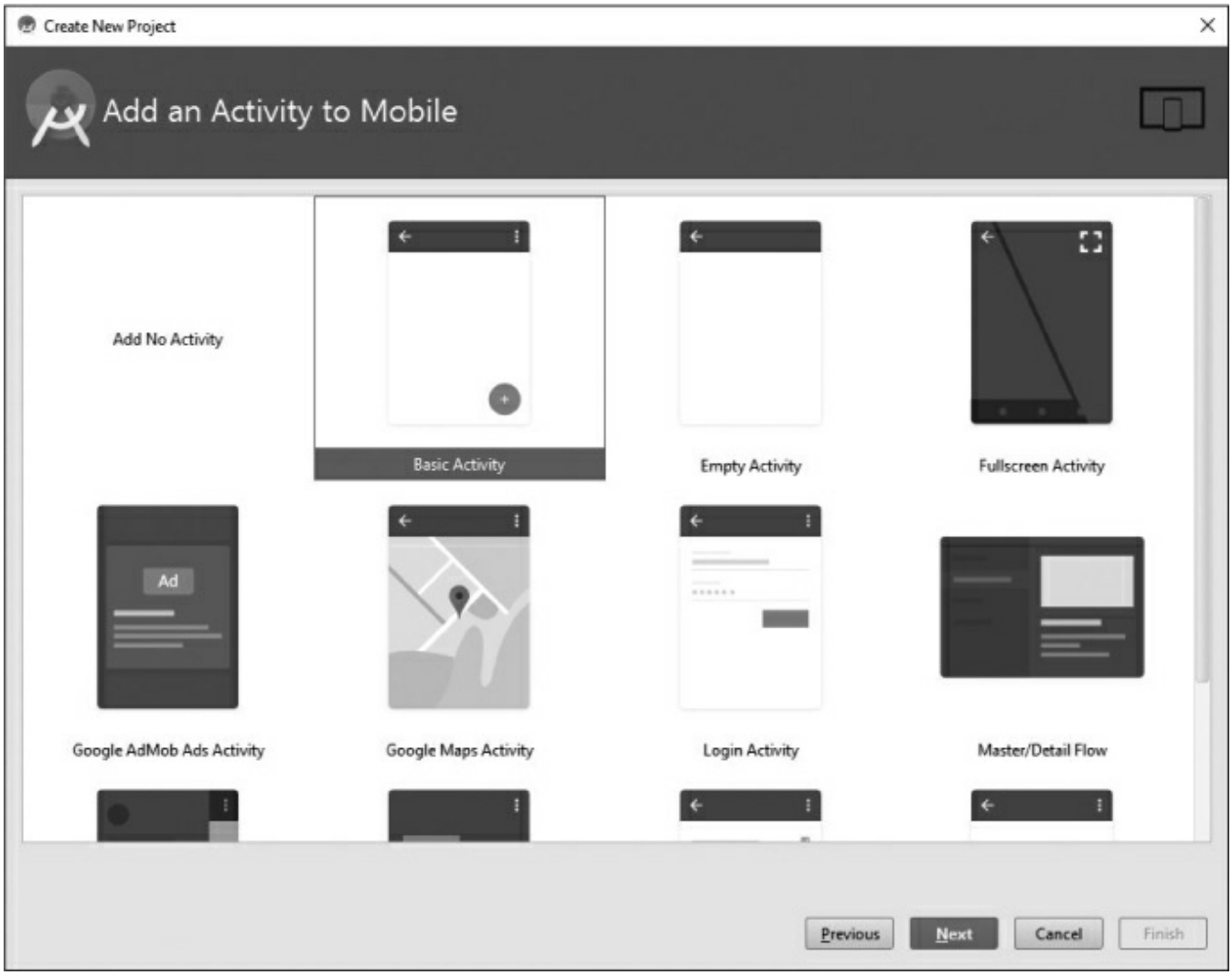


Figure 4-5: Choose Basic Activity as the basis for your app.

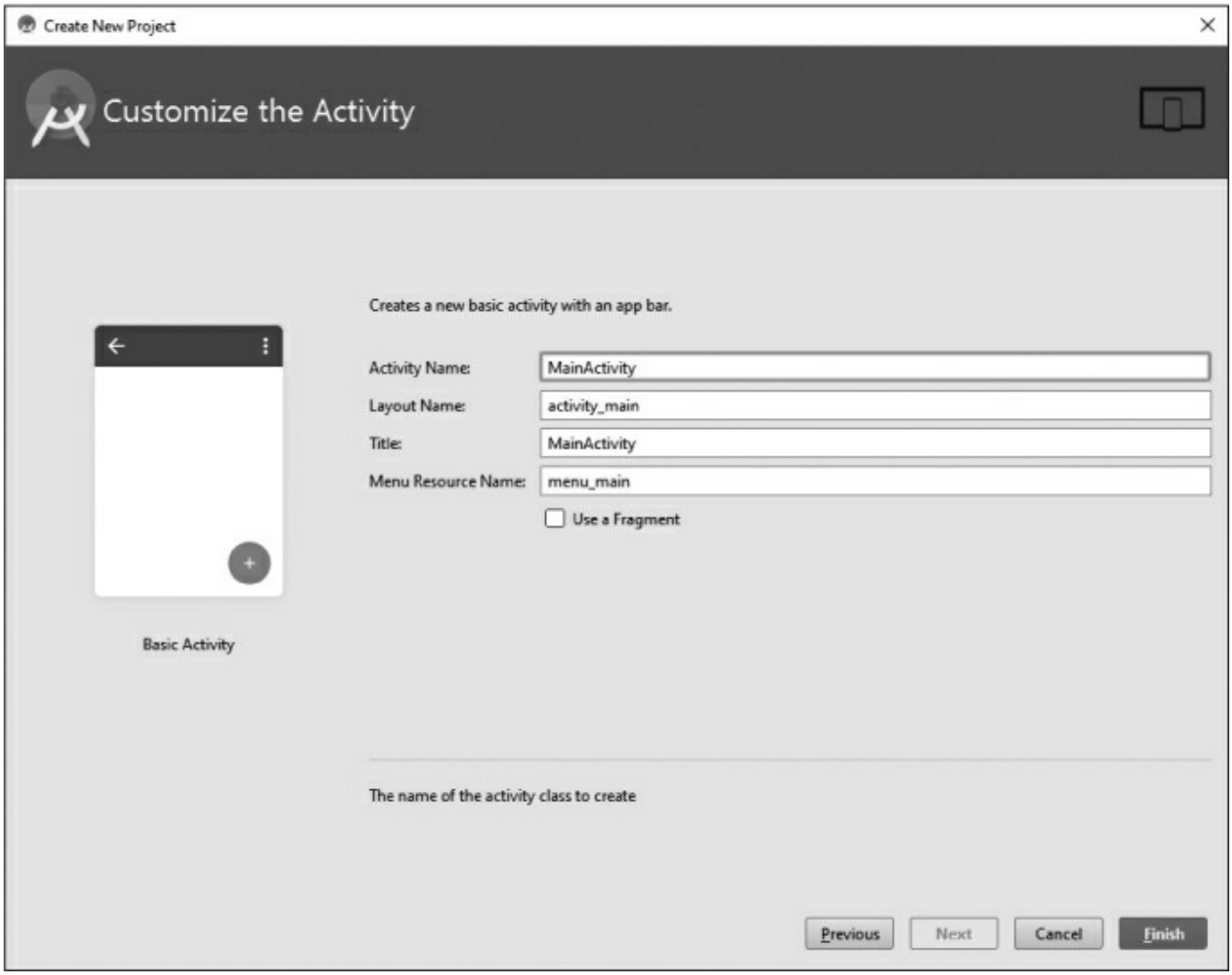


Figure 4-6: Leave the default values for activity name and layout name.

The project may take a few moments to build, as Android Studio creates the project files and sets up the *GuessingGame* app project. After it finishes loading, Android Studio will open the project to a default view that looks like Figure 4-7.

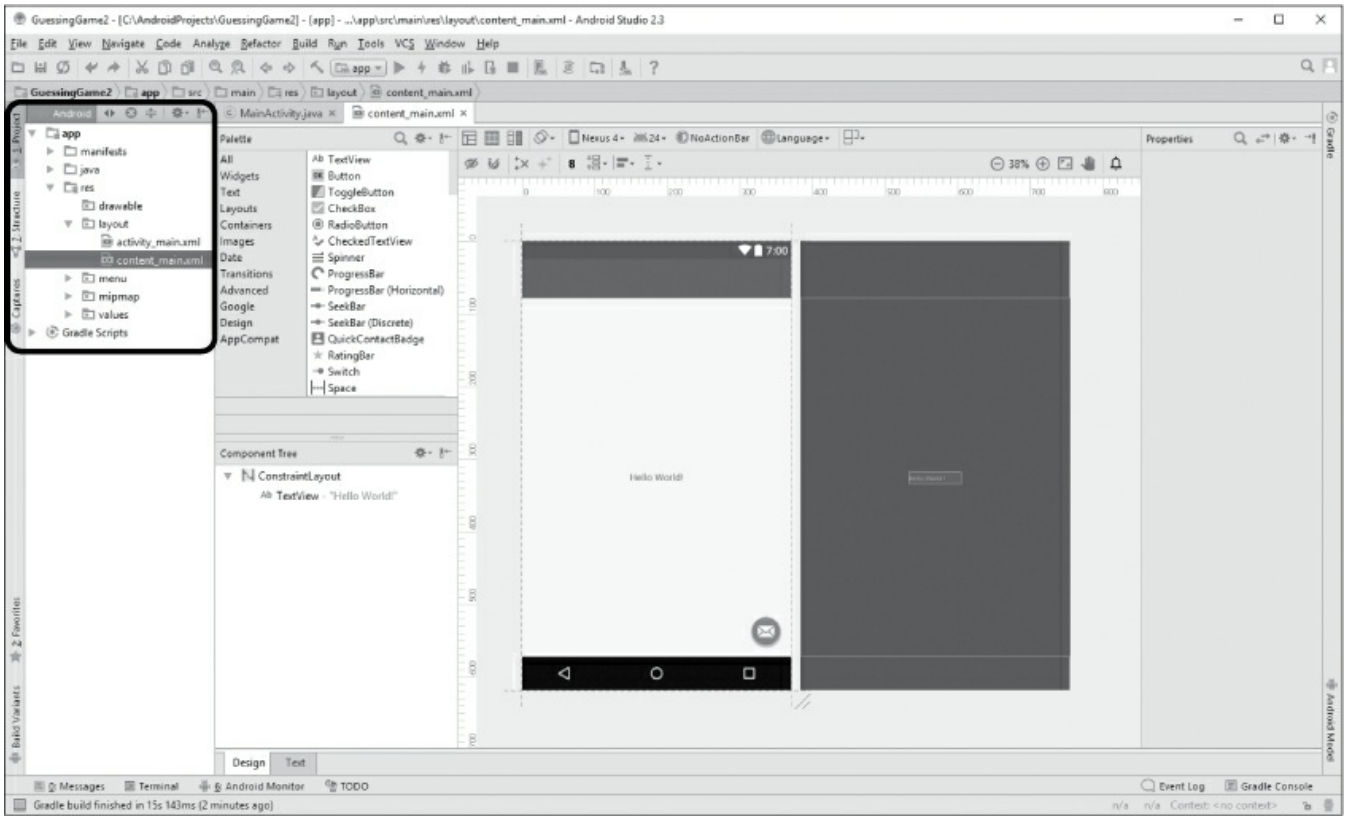


Figure 4-7: The default project view in Android Studio, complete with a GUI mobile layout view

If your project doesn't open with this screen, click the Project tab in the upper left, expand the *app* folder in the Project Explorer, expand the *res* (short for *resources*) folder, expand *layout*, and then double-click the *content_main.xml* file to see a view similar to Figure 4-7.

Building the GUI Layout in Design View

Android apps separate the layout from the app code (the activity), so this app will be a little different from the desktop app we wrote in Chapter 3. The layout isn't described in Java but in *eXtensible Markup Language (XML)*. Fortunately, despite this difference, we can still use a drag-and-drop palette similar to Eclipse's WindowBuilder Editor to design the GUI view. The main difference in Android Studio is that the components have been named with mobile apps in mind.

Like the WindowBuilder Editor, Android Studio has two tabs, one for the design view and one for the source code. Click the Design tab in the lower-left corner of the main window containing the *content_main.xml* file. You'll see the design view shown in Figure 4-8.

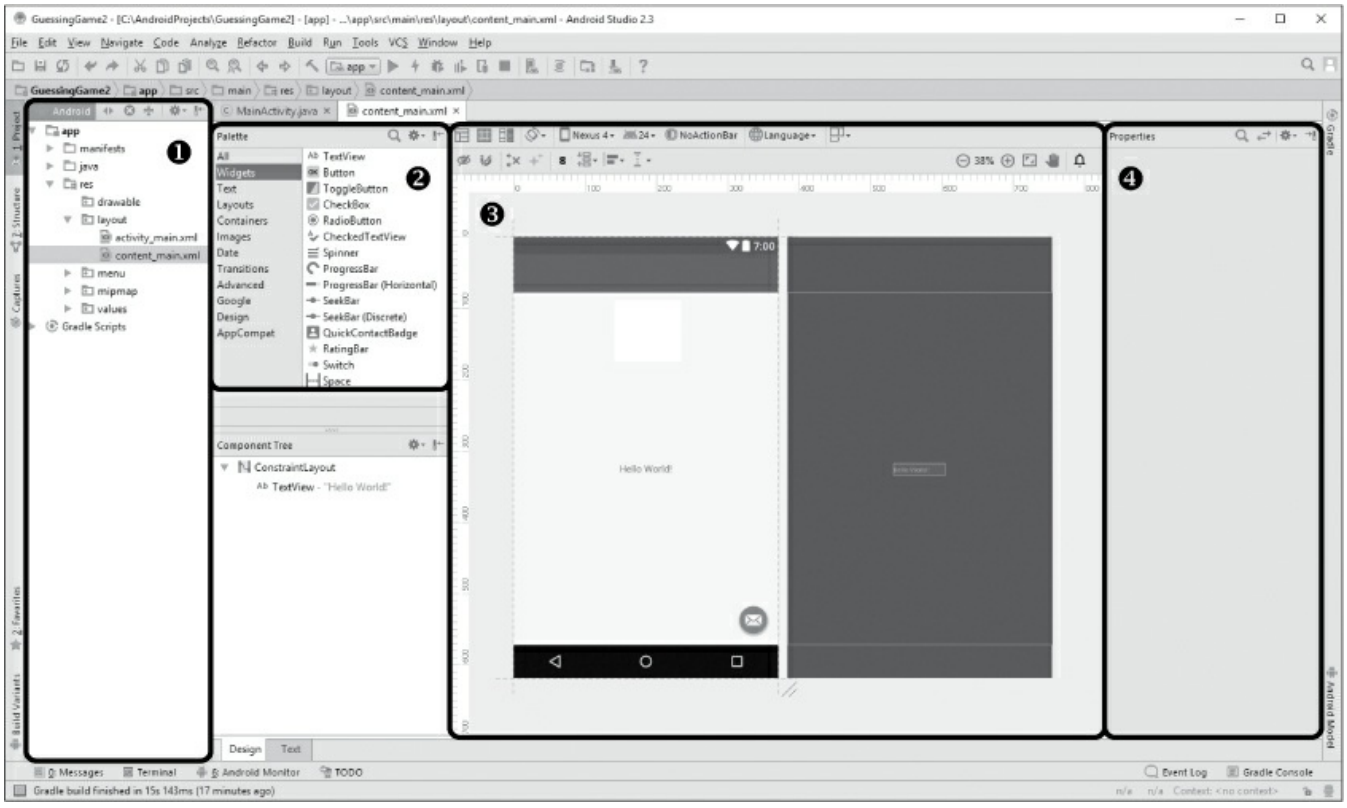


Figure 4-8: Android Studio has a design view similar to the WindowBuilder Editor in Eclipse.

Figure 4-8 shows the four areas of the design view that we'll be using the most: the Project Explorer pane ❶, the Palette pane ❷, the Preview pane ❸, and the Properties pane ❹.

In the center of the Preview pane, you'll see a label—known as a `TextView` in Android terminology—reading *Hello World!*. Click the label and press DELETE (or right-click and select **Delete** from the menu) to delete it from the Preview pane. Then, click the **Layouts** option in the Palette pane, select a **RelativeLayout**, and drag it either to the Preview pane or onto the ConstraintLayout (in Figure 4-8) in the Component Tree listing just below the Palette pane. We'll start with an empty RelativeLayout to build our guessing game GUI view.

In the Palette pane, under Widgets, select **TextView**. A `TextView` in Android is similar to a `JLabel` in the Swing toolkit, and *widgets* are what we call most of the GUI components in Android. We'll drag and drop to place GUI components in the design preview.

Click the `TextView` widget and drag it over the white app background in the simulated Android phone, placing the widget near the top to serve as the title of your app.

After placing the title `TextView`, you'll see options appear in the Properties pane on the right. If the pane doesn't appear, click Properties in the upper-right corner of the Design tab to expand the Properties pane. Find the `text` property of the `TextView` widget you've just added. Change the text to **`"Your Name's Guessing Game"`** and change the `textAppearance` property to **Large**. Then, drag the `TextView` to center it at the top of the screen as shown in Figure 4-9. The gray dashed guideline helps with placement.

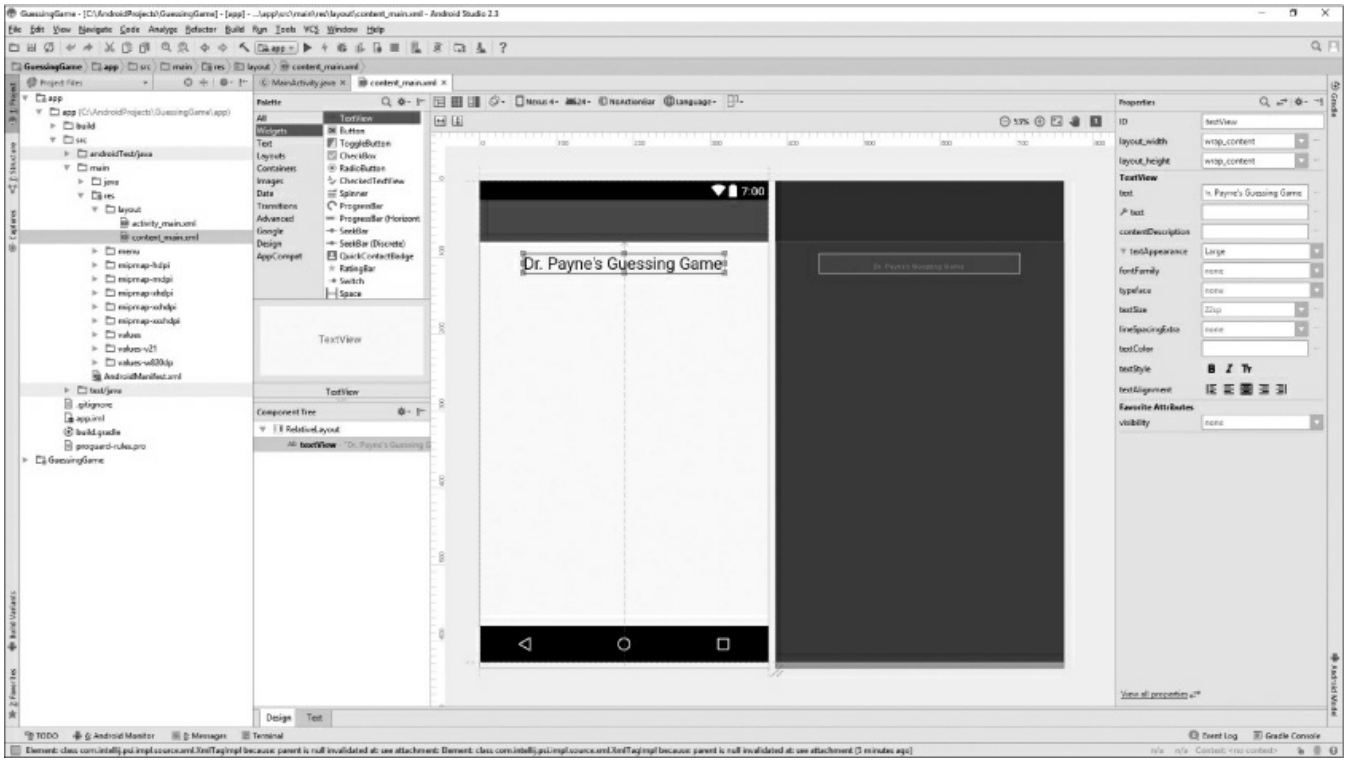


Figure 4-9: Placing GUI widgets/components in Android Studio is very similar to the process in Eclipse.

To add the label prompting the user to enter a number between 1 and 100, place another `TextView` widget onto the design preview just a small distance below the title. Change the `textAppearance` property to **Medium** in the Properties pane on the right. Then, type **`Enter a number between 1 and 100:`** in the `text` property for the `TextView`.

Next, we'll place the text field, also known as an `EditText` in Android, for the user's guess. Text fields are listed just below Widgets in the Palette pane. There are many types of text fields in Android, and each behaves differently depending on the needs of your app. For example, Plain Text is a simple text field, Password hides characters as they're typed, Phone shows the numeric keypad and formats the input as a phone number, and E-mail shows a modified keyboard with the @ symbol.

Choose a **Number** text field so the user can enter their guess from a numeric keyboard on their screen. Place this field just a little below the previous label. Then, place a Button widget an equal distance below the text field and change its `text` property to **`"Guess!"`**. Finally, place another `TextView` widget below the button, giving it a `text` property of **`"Enter a number, then click Guess!"`**. If you have trouble placing the widget the same distance below the button because you're too close to the vertical center of the screen, place it lower or higher in the Properties pane under Layout_Margin (you may need to click **View all properties** and then click the gray arrow next to Layout_Margin to expand the margin properties), find **layout_marginTop**, and change the value to **30dp**. The finished GUI layout should look similar to Figure 4-10.

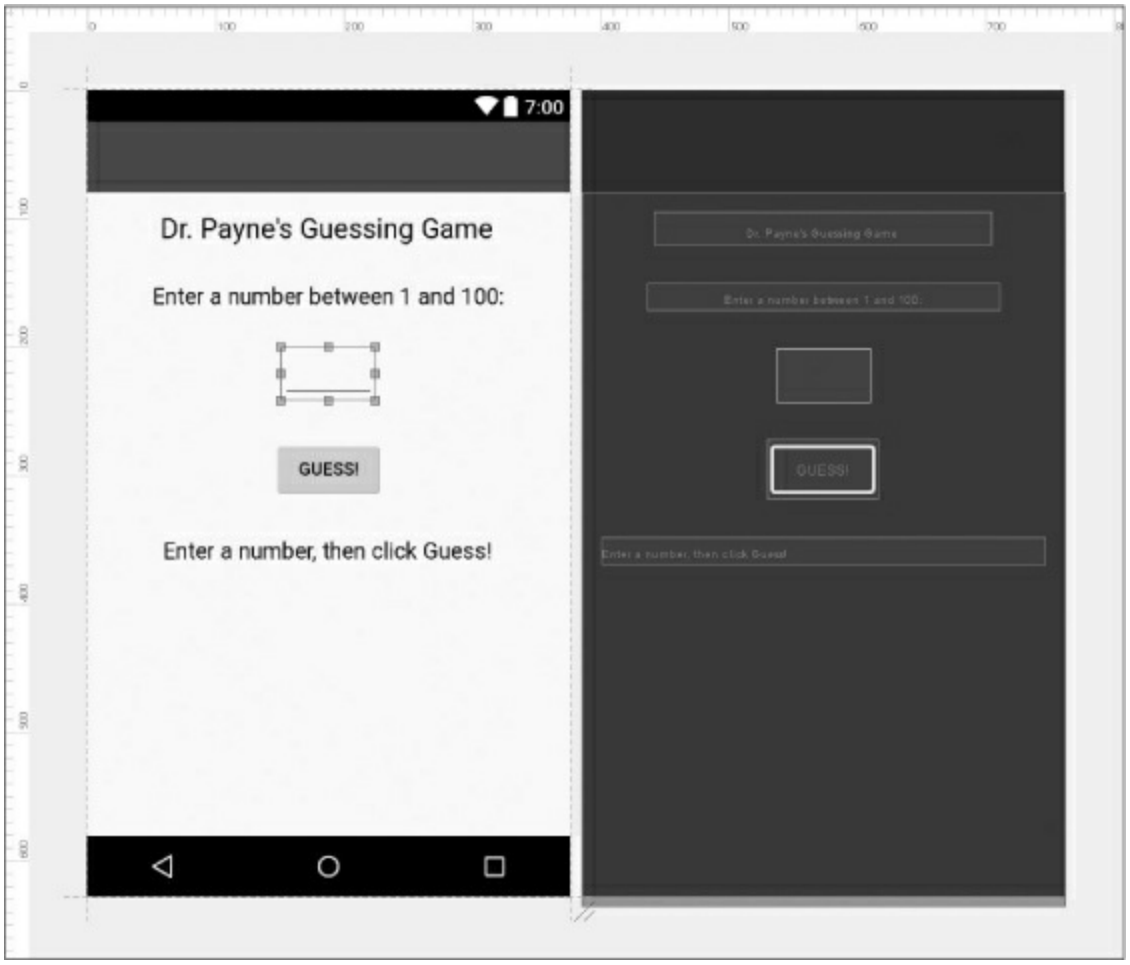


Figure 4-10: The finished GUI layout for the Android guessing game app in design view

We'll make one final change before naming the GUI components. Change the `width` property of the numeric text field, where the user enters their guess, to **75dp**; do so by clicking **View all properties** or the left-and-right arrow icon in the Properties pane. The original text field was too wide for a number-guessing

game that uses only numbers between 1 and 100, so we've resized it to make it narrower.

With that small change, it's time to start naming our GUI components so we'll be able to find them easily in the Java code.

Naming GUI Components in Android Studio

Just as we named GUI components in Eclipse, we'll need to name the GUI elements in Android Studio so that we can access them from the code. But in Android Studio, the names of the GUI elements themselves won't appear in the Java source code by default. Instead, we'll have to connect these XML GUI components to the Java code by hand before we can use them in the code.

For now, though, we just need to change the `id` property of the number text field, the guess button, and the bottommost label to **`txtGuess`**, **`btnGuess`**, and **`lblOutput`**, respectively. If a pop-up message asks you to update usages, click **Yes**. We'll use these names for consistency and convenience. Figure 4-11 shows the renamed text field, button, and label.

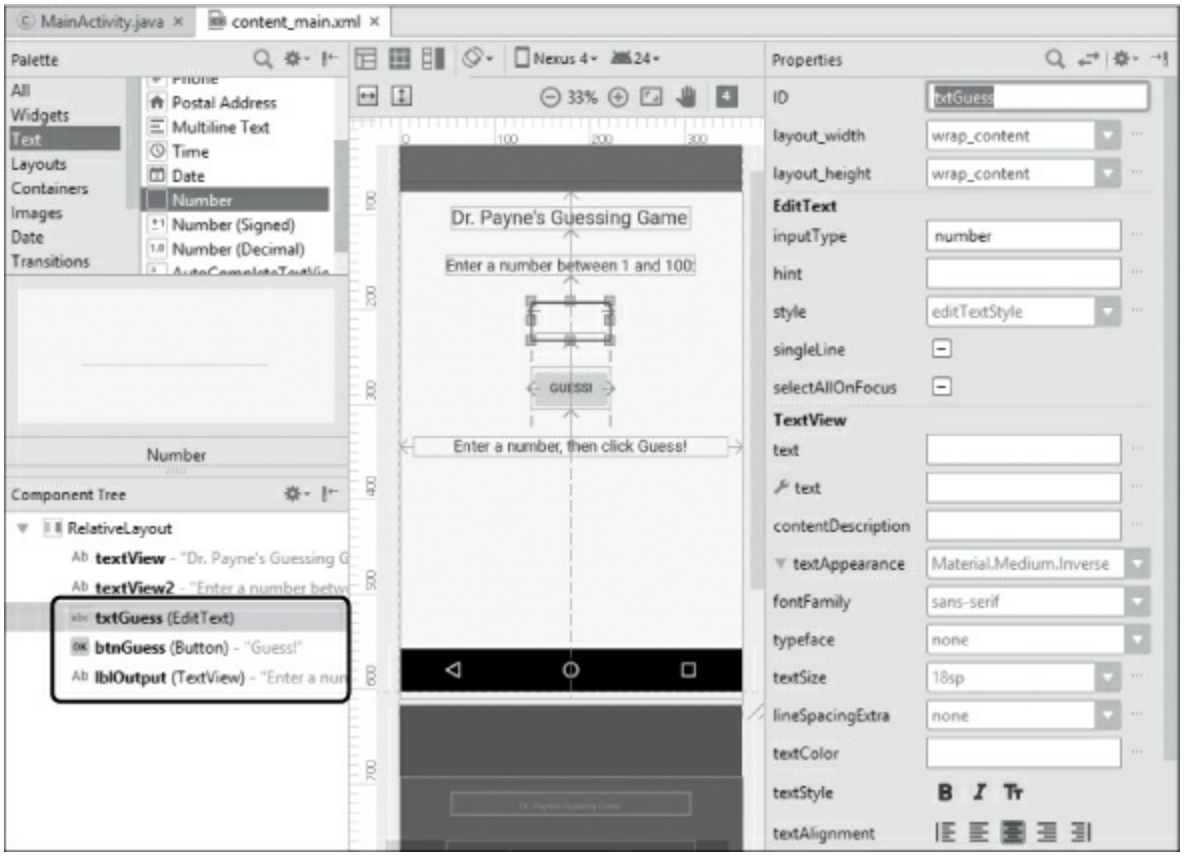


Figure 4-11: Rename the `EditText`, `Button`, and `TextView` components (circled) by changing their id

properties to `txtGuess`, `btnGuess`, *and* `lblOutput`, *respectively.*

One last thing we should do before we start coding in Java is hide the small *floating action button (fab)* icon, shown in the bottom right of the design preview in Figure 4-12. Your project may not show the fab icon, but if it does, double-click the *activity_main.xml* file in the Project Explorer pane ❶. Click the *activity_main.xml* tab above the design view ❷ and then click the fab icon ❸. Go to Properties and set the `visibility` property to **invisible** ❹.

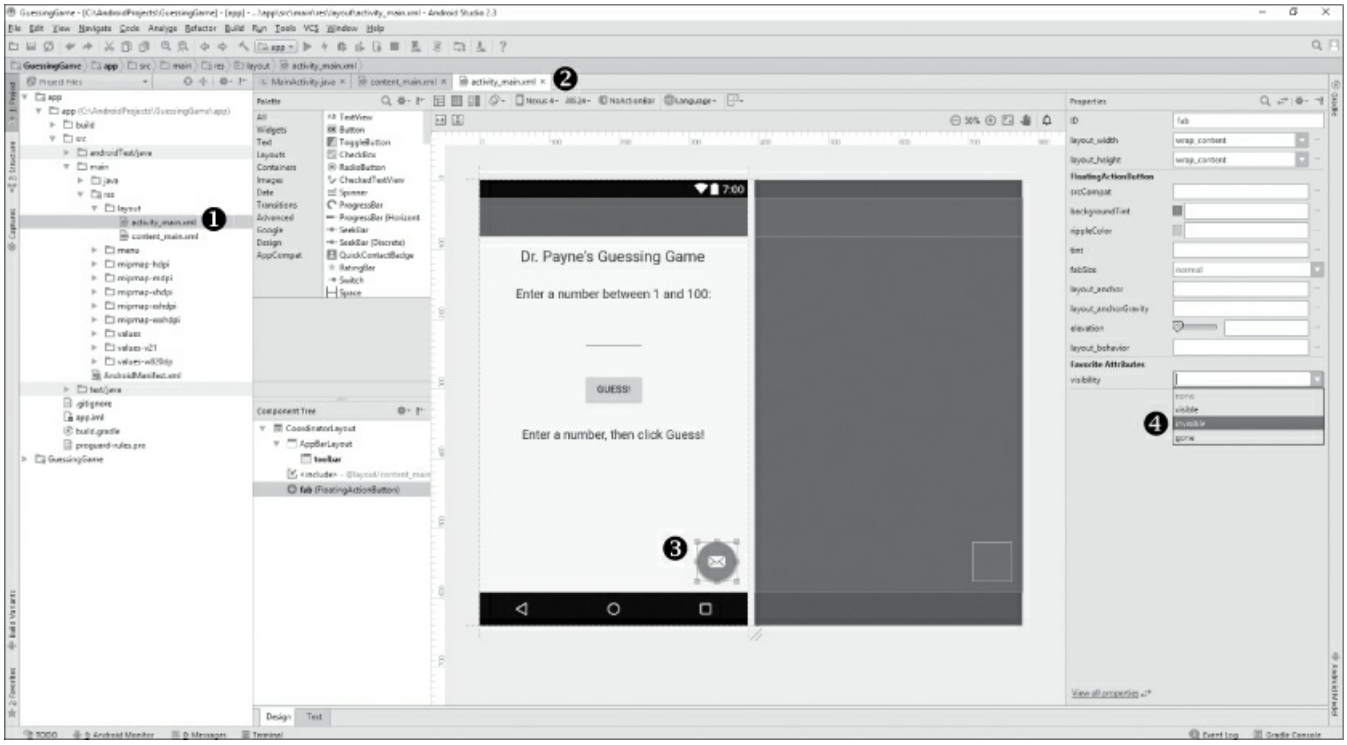


Figure 4-12: Hide the fab icon by going to the activity_main.xml *file, clicking the small envelope icon, and setting the* `visibility` *property to invisible.*

I've chosen to set the fab icon to invisible rather than delete it entirely, because you might want to use the floating action button to add new features to your app later on. The floating action button can share info from the app to Facebook, post a tweet in Twitter about how much you like the app, send an email to a friend about the app, and more.

Next, we'll wire up the GUI so that we can use it in Java to program the rest of the app.

Connecting the GUI to Java in Android Studio

It's time to connect the GUI to Java so we can write the code for the guessing game app. First, let's open the Java source code file by looking in the Project Explorer pane, under *app* ▸ *src* ▸ *main* ▸ *java* ▸ *com.example.guessinggame* (or your package name) ▸ *MainActivity*. Double-click *MainActivity* to open the *MainActivity.java* source code file.

You may have noticed the *java* folder in the Project Explorer has another package or two, possibly labeled the same, but with *(test)* or *(androidTest)* after the package name. These packages are used for testing in larger applications, where engineers evaluate app quality, security, and functionality for software quality assurance. We won't tackle quality assurance, but it's a great way to get into the software development industry.

The Java source code inside the *MainActivity.java* file will look something like Figure 4-13.

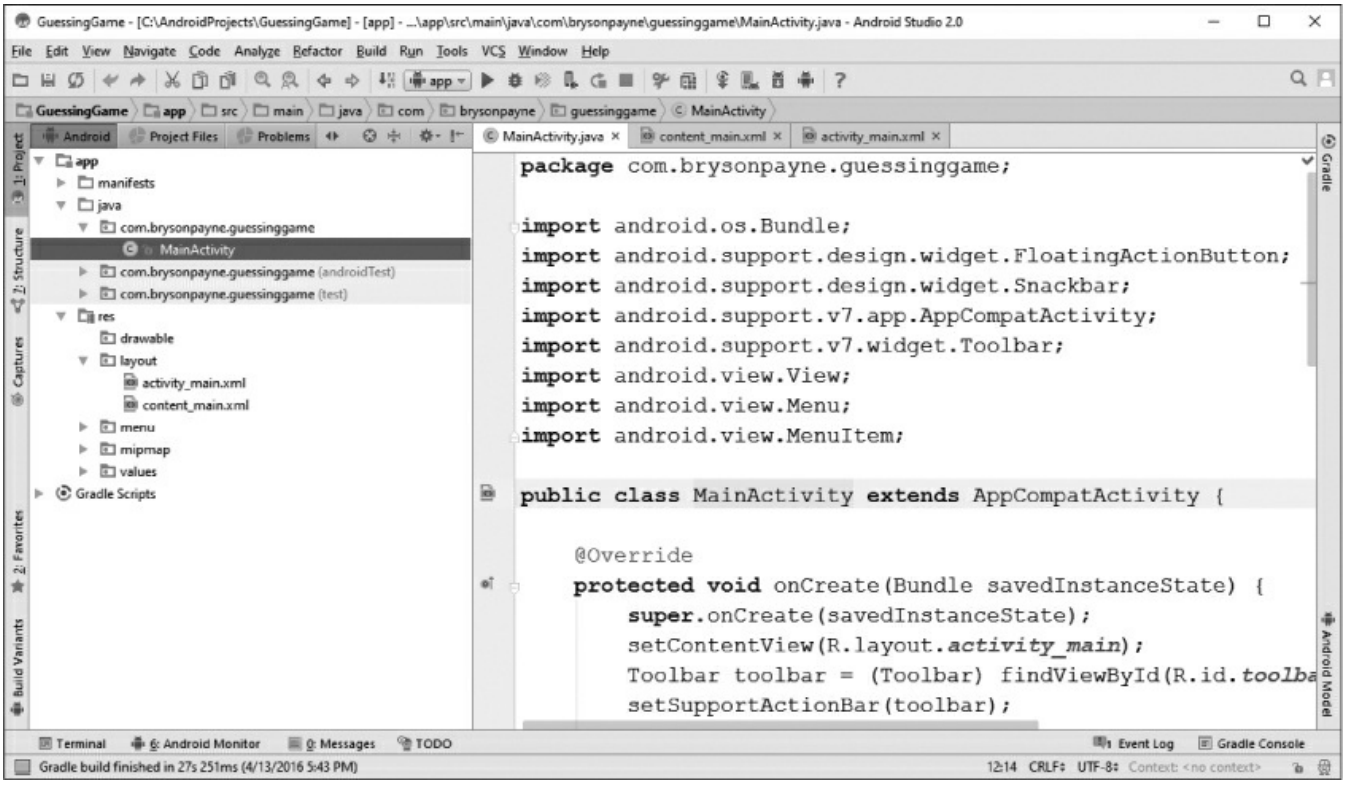


Figure 4-13: The Java source code for the app is called MainActivity.java *by default.*

Notice that in Android Studio, a package declaration appears at the very top of the file by default. Packages help us organize all the files needed in a more complex program, such as a mobile app. In this case, the package name is the company domain name from the Create New Project screen back in Figure 4-3 on page 77, but in reverse order—notice the *com* comes first.

Several Android `import` statements follow the package declaration. These work just as they do for Java on the desktop, importing existing features and functionality into the code.

The following `public class MainActivity` code snippet may be slightly different for you depending on the minimum API level you chose for your app, but the overall code will be similar, and the app we're writing will work across multiple API levels. First we'll declare our variables to connect the GUI to the program. Click just inside the opening brace for the class, on the next line down, and add these lines to declare variables for the text field, button, and output label:

```
public class MainActivity extends AppCompatActivity {
    private EditText txtGuess;
    private Button btnGuess;
    private TextView lblOutput;
```

As you enter each new variable type, you may see a drop-down menu offering to import it (`android.widget.EditText`, for example). If you click the correct widget type to import, Android Studio will add the corresponding `import` statement for you. If you typed the three lines without clicking to accept the automatic imports, just click each widget type and then press ALT-ENTER (or OPTION-ENTER on macOS) to import any missing classes, as shown in Figure 4-14.

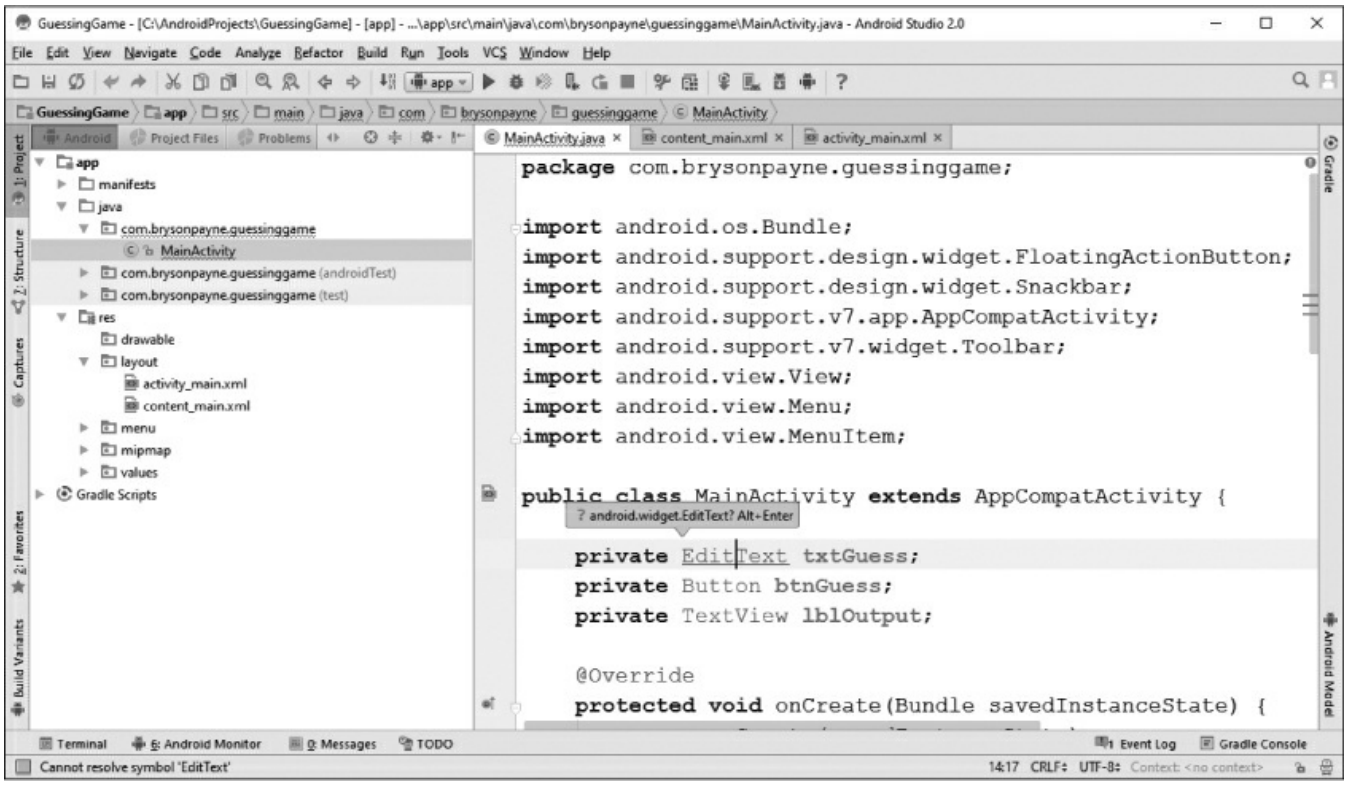


Figure 4-14: Android Studio enables you to automatically import classes just as Eclipse did, either as you type or afterward with a simple key combination (ALT-ENTER, or OPTION-ENTER on macOS).

The following three `import` statements should now appear with the others near the top of your file:

```
import android.widget.EditText;
import android.widget.Button;
import android.widget.TextView;
```

After declaring the three variables for the GUI widgets, we need to connect those variables to the XML components for each one. We'll do this in the `onCreate()` method, shown at the bottom of Figure 4-14, which is the function that runs when the app is loaded. Android Studio has automatically generated the code to begin this method.

Look inside the top two or three lines of the `onCreate()` method, and find this line:

```
    setContentView(R.layout.activity_main);
```

Right after that line, press ENTER a couple of times and type the following partial line of code to begin connecting the variable `txtGuess` to the `EditText` widget in the XML layout file:

```
    txtGuess = (EditText) findViewById(R.id.
```

The `findViewById()` function is how we connect GUI widgets in the XML layout to the variables we'll use to represent them in the source code. The `R` inside that function refers to a special file called *R.java* that Android Studio generates to make it possible to connect resources. The `R` is short for *resources*, usually stored in the *res* folder in the project. As you begin to type the preceding line, you'll see a pop-up menu like the one shown in Figure 4-15. Find and double-click `txtGuess` in the drop-down list of options. If you don't see the `txtGuess` option, go back to the design view for *content _main.xml* and make sure you gave the text field the `id` property of `txtGuess`.

Finish with the closing parenthesis followed by a semicolon, as shown in the following listing, and then set up the button and output label. The finished three lines of code inside your `onCreate()` method should look like this:

```
protected void onCreate(Bundle savedInstanceState) {
```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
txtGuess = (EditText) findViewById(R.id.txtGuess);
btnGuess = (Button) findViewById(R.id.btnGuess);
lblOutput = (TextView) findViewById(R.id.lblOutput);
```

If everything has been named correctly in the layout, these lines of code should connect the variables `txtGuess`, `btnGuess`, and `lblOutput` to the `EditText`, `Button`, and `TextView` components in the GUI layout. This is a good time to stop and save your project to capture the progress you've made so far.

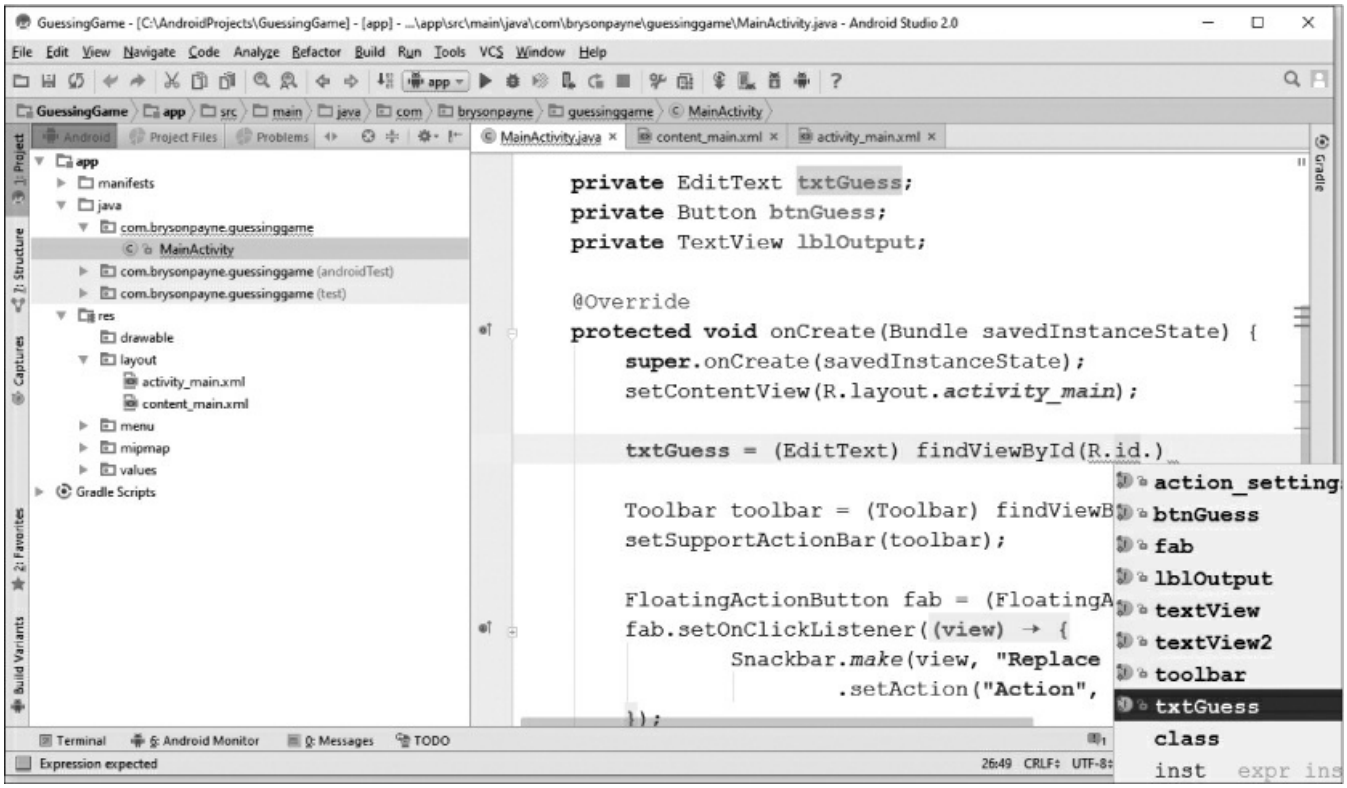


Figure 4-15: Android Studio helps us "wire up" the Java code to the GUI resources in the layout with helpful drop-down options.

Adding Methods to Check the Guess and Begin a New Game

Now, let's work on writing the code for the `checkGuess()` function. We can begin writing the `checkGuess()` method just below the variable declarations for `txtGuess`, `btnGuess`, and `lblOutput` and just above the `onCreate()` method, as shown in Figure 4-16.

First, we need to get the user's guess from the GUI text field and store it in a

`String` variable:

```
public class MainActivity extends AppCompatActivity {
    private EditText txtGuess;
    private Button btnGuess;
    private TextView lblOutput;
    public void checkGuess() {
        String guessText = txtGuess.getText().toString();
```

The code to get the user's guess from the text field looks almost the same as in the desktop version of the app, except for the additional `.toString()` method at the end. There is a separate `Text` class in Android and the text that is entered into a text field is a `Text` object, so we have to convert that object to a `String` object.

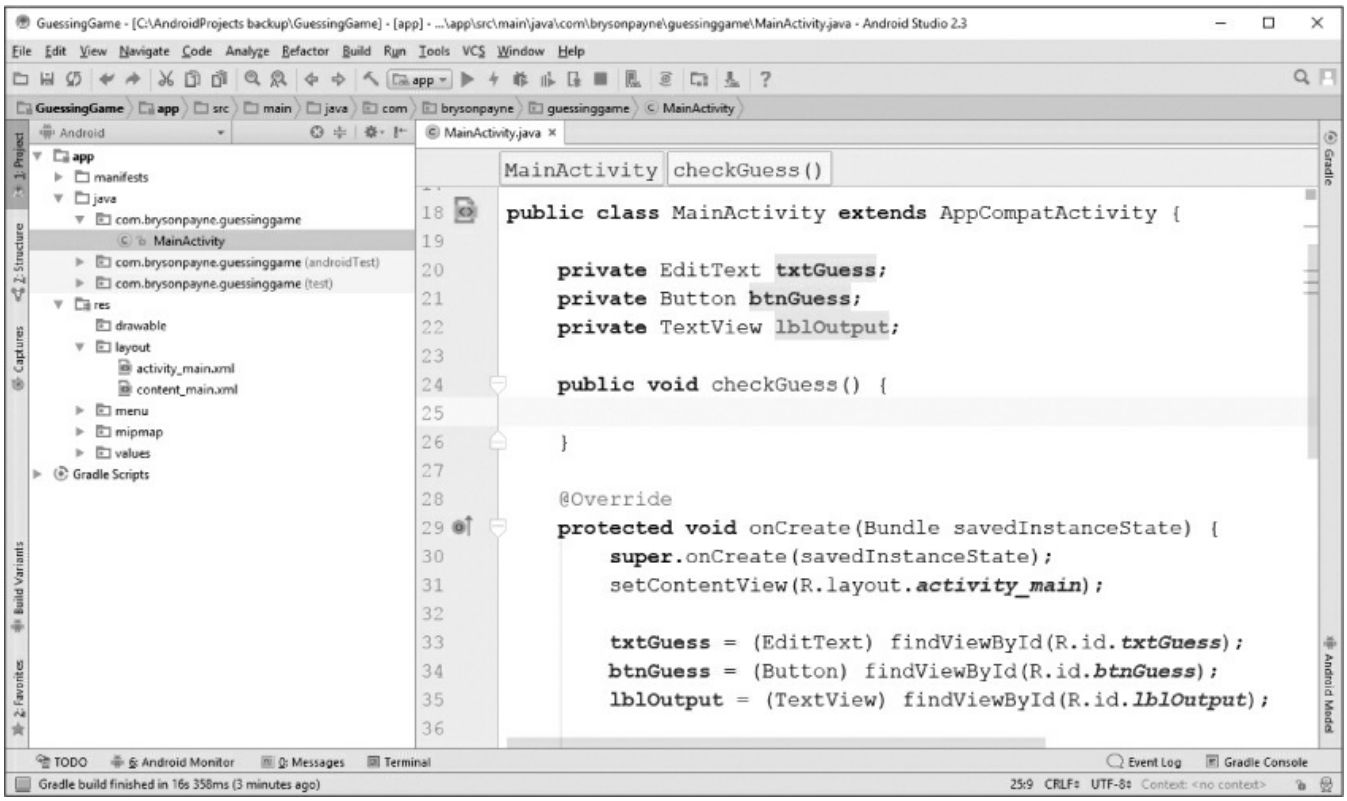

GuessingGame - [C:\AndroidProjects backup\GuessingGame] - [app] - ...\app\src\main\java\com\brysonpayne\guessinggame\MainActivity.java - Android Studio 2.3
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MainActivity.java
MainActivity
checkGuess()
public class MainActivity extends AppCompatActivity {
private EditText txtGuess;
private Button btnGuess;
private TextView lblOutput;
public void checkGuess() {
}
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
txtGuess = (EditText) findViewById(R.id.txtGuess);
btnGuess = (Button) findViewById(R.id.btnGuess);
lblOutput = (TextView) findViewById(R.id.lblOutput);
Gradle build finished in 16s 358ms (3 minutes ago)

Figure 4-16: Beginning to write the checkGuess() method in Java

Fortunately, the rest of the `checkGuess()` method can be copied and pasted directly from the Eclipse desktop GUI version of the app from Chapter 3, with no changes! The complete `checkGuess()` method should look like this:

```
public void checkGuess() {
  ❶ String guessText = txtGuess.getText().toString();
    String message = "";
  ❷ try {
      ❸ int guess = Integer.parseInt(guessText);
```

```
        if (guess < theNumber)
            message = guess + " is too low. Try again.";
        else if (guess > theNumber)
            message = guess + " is too high. Try again.";
        else {
            message = guess +
                " is correct. You win! Let's play again!";
            newGame();
        }
  ❹ } catch (Exception e) {
        message = "Enter a whole number between 1 and 100.";
  ❺ } finally {
        lblOutput.setText(message);
        txtGuess.requestFocus();
        txtGuess.selectAll();
    }
}
```

One of the convenient features of Java is that we can reuse code across platforms. Just as we used code from the command line game to make the desktop game, we can use code from the desktop version in the Android version. Let's review this code to understand how it will work in the Android app.

At ❶, we get the user's guess from the text field and set up a string for the output message. At ❷, we begin a `try-catch-finally` statement to handle user input errors or exceptions. Inside the `try` block, at ❸, we parse the integer number of the user's guess from the characters they enter. The rest of the `try` block contains the `if-else` statements that test whether the user guessed too high or too low, set the appropriate message, and start a new game. Next, the `catch` statement ❹ that tells the user to enter a whole number between 1 and 100. Then, the `finally` block ❺ sets the text on the `lblOutput` widget to the appropriate `message`, returns the cursor to the text field, and highlights the text to prepare for the user's next guess.

You may have noticed that there are a couple of items underlined in red: `theNumber` and `newGame()`. This is because we haven't defined them yet in this version of our app.

At the top of the `MainActivity` class, below the three GUI widget declarations, add a declaration for the secret number, `theNumber`:

```
public class MainActivity extends AppCompatActivity {
    private EditText txtGuess;
    private Button btnGuess;
    private TextView lblOutput;
    private int theNumber;
```

This is the same code we used in the desktop game, because the Java code to create an integer is the same across the command line, desktop, and Android mobile versions of the app.

The code for the `newGame()` method is also the same as the desktop version. Add the `newGame()` method just before the `onCreate()` statement:

```
    public void newGame() {
        theNumber = (int)(Math.random() * 100 + 1);
    }
@Override
protected void onCreate(Bundle savedInstanceState) {
```

Creating a new game in the Android app is just like doing so in the desktop app. We just set `theNumber` to a random integer from 1 to 100, using the `Math.random()` function.

We can add a call to `newGame()` inside the `onCreate()` method after the code that connects the three GUI components:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    txtGuess = (EditText) findViewById(R.id.txtGuess);
    btnGuess = (Button) findViewById(R.id.btnGuess);
    lblOutput = (TextView) findViewById(R.id.lblOutput);
    newGame();
```

When the game starts up, the program will call the `newGame()` method and pick a random number for the user to guess. Now, we just need to handle the button click event and learn how to run the app on our own Android device or on the Android emulator.

Handling Events in Android

In Eclipse, we were able to add an event listener to the guess button just by double-clicking it in the design view. Unfortunately, it's not that easy in Android Studio, because the GUI layout is separate from the source code. Fortunately, Android Studio helps us add event listeners by providing *code completion* options, similar to the content assist feature in Eclipse.

To add a listener for button clicks, begin typing the following partial line of

code inside the `onCreate()` method on the next line after the call to `newGame()`:

```
    txtGuess = (EditText) findViewById(R.id.txtGuess);
    btnGuess = (Button) findViewById(R.id.btnGuess);
    lblOutput = (TextView) findViewById(R.id.lblOutput);
    newGame();
    btnGuess.setOn
```

Android Studio's code completion feature will pop up with a list of code recommendations, as shown in Figure 4-17. Choose `setOnClickListener()` from the list of options and double-click to add it to your program.

Inside the parentheses for `btnGuess.setOnClickListener()`, type **`new`** and start typing **`OnClickListener`**. Android Studio's code completion feature will display a list of options, as shown in Figure 4-18.

Choose `OnClickListener` from the list of options, and you'll see Android Studio adds a few extra lines of code. Your event listener code for the `btnGuess` button should now look like the following:

```
btnGuess.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

Look familiar? This is another example of an anonymous inner class. The desktop version also had an anonymous inner class, but it was named slightly differently. The two will work virtually identically. You may have noticed that Android inserts `@Override` in several places. This is called a *compiler directive*, and it tells the compiler that you're implementing your own version of a method in the parent class.

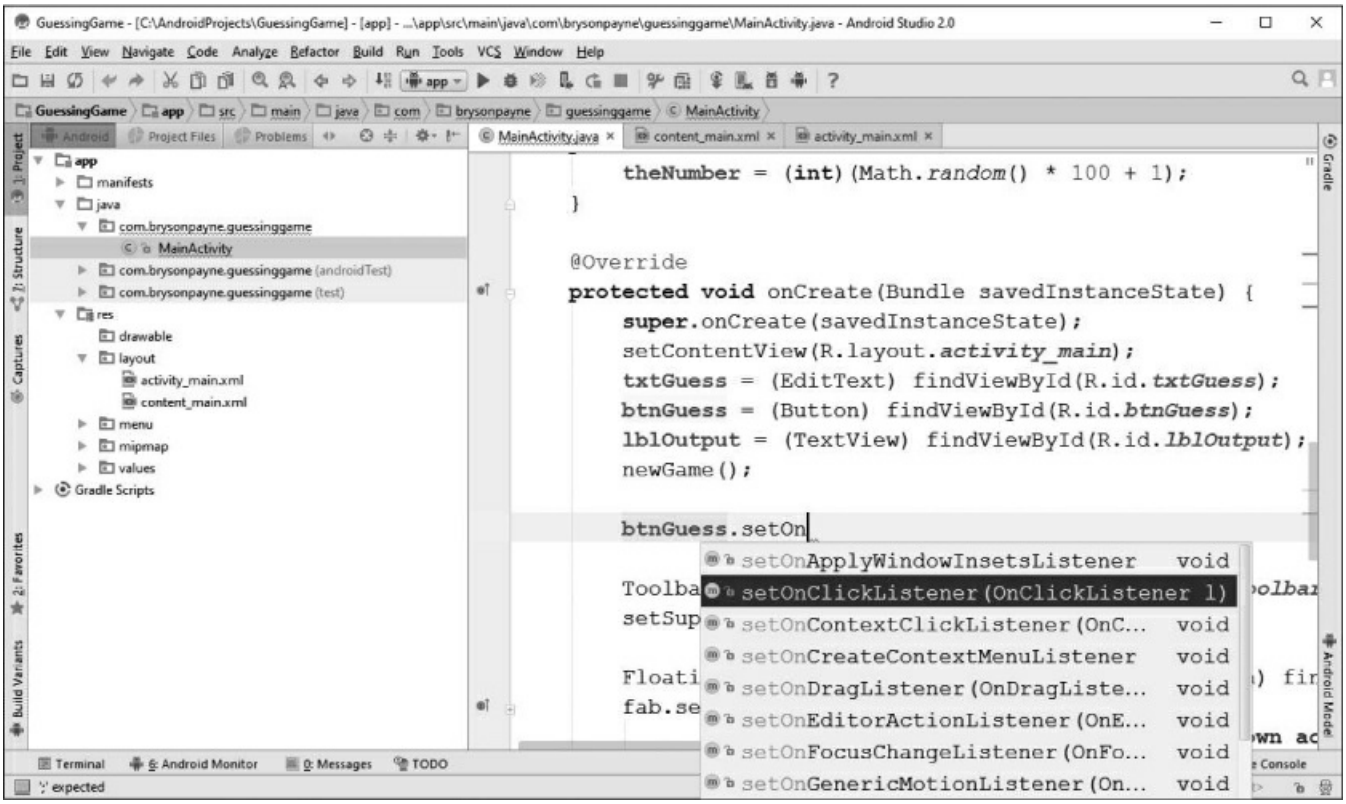


Figure 4-17: Android Studio's code completion feature suggests code as we type, similar to Eclipse's content assist feature.

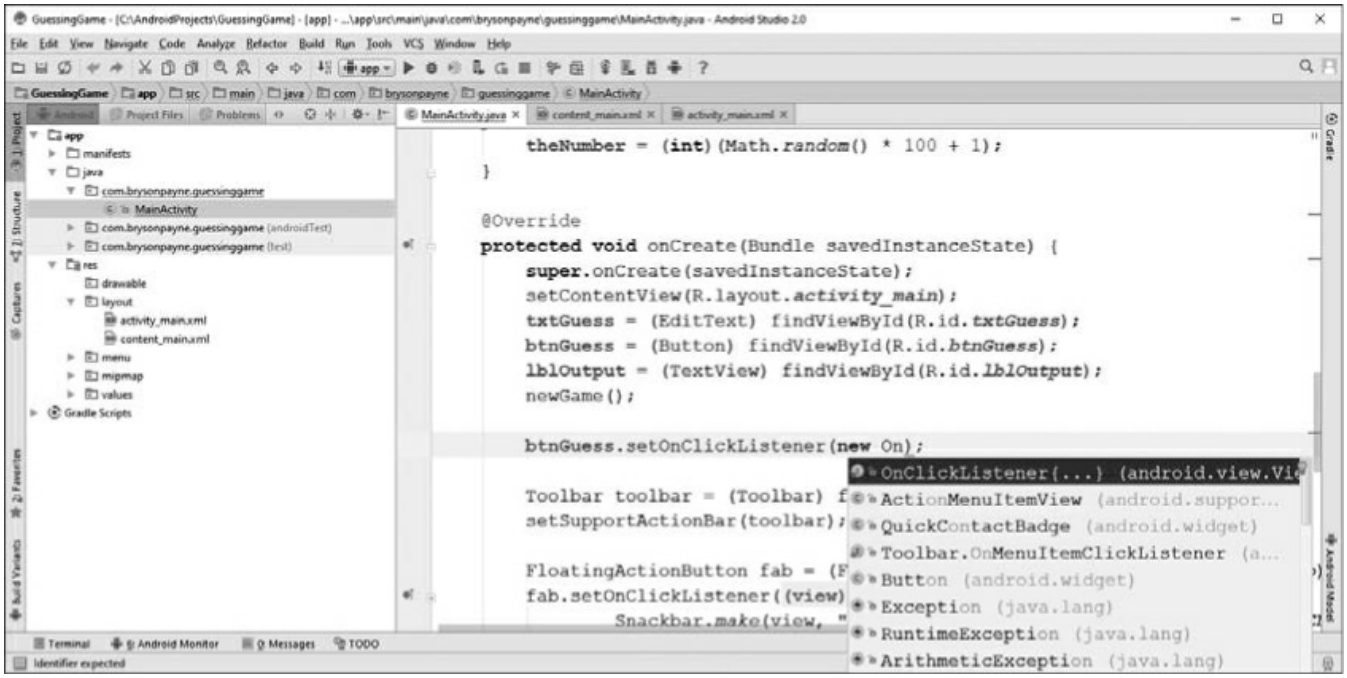


Figure 4-18: Use the code completion feature to create a new OnClickListener to add to the button btnGuess.

When the user clicks the Guess! button, we need to check their guess against the secret number. Add `checkGuess();` inside the braces of the `onClick()` method.

```
public void onClick(View v) {
    checkGuess();
}
```

The guessing game mobile app is now ready for its first run. The fully functional code for this first version of the guessing game is shown here. Your version may differ slightly, perhaps with a couple of extra methods for handling menu items, but because we're not using the menu, these have been removed for space (see Chapter 5 to learn how to create options menus and settings):

```
package com.brysonpayne.guessinggame;
import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.EditText;
import android.widget.Button;
import android.widget.TextView;
import org.w3c.dom.Text;
public class MainActivity extends AppCompatActivity {
    private EditText txtGuess;
    private Button btnGuess;
    private TextView lblOutput;
    private int theNumber;
    public void checkGuess() {
        String guessText = txtGuess.getText().toString();
        String message = "";
        try {
            int guess = Integer.parseInt(guessText);
            if (guess < theNumber)
                message = guess + " is too low. Try again.";
            else if (guess > theNumber)
                message = guess + " is too high. Try again.";
            else {
                message = guess +
                      " is correct. You win! Let's play again!";
                newGame();
            }
       } catch (Exception e) {
           message = "Enter a whole number between 1 and 100.";
       } finally {
           lblOutput.setText(message);
           txtGuess.requestFocus();
           txtGuess.selectAll();
       }
   }
   public void newGame() {
```

```
        theNumber = (int)(Math.random() * 100 + 1);
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txtGuess = (EditText) findViewById(R.id.txtGuess);
        btnGuess = (Button) findViewById(R.id.btnGuess);
        lblOutput = (TextView) findViewById(R.id.lblOutput);
        newGame();
        btnGuess.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                checkGuess();
            }
        });
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

In the next two sections, you'll learn how to run the app on the Android emulator and on real Android devices.

Running the App on the Android Emulator

Now we have a working version of the Java code to run the app, but we still need to test it. Unlike the command line and desktop apps, the mobile app can't run on a PC by itself, because PCs don't have the Android operating system. To run your app for testing, you'll need either an Android device or an emulator that will simulate an Android device on your PC. In this section, we'll create an Android virtual device for testing your apps.

While viewing the *MainActivity.java* file, press the run button or select the menu item **Run ▸ Run 'app'**. A pop-up window will ask you to select a *deployment target*, which is a device you want to run the app on, as shown in Figure 4-19.

Figure 4-19: To run an Android app, you must choose a target device—either an emulator or an actual device.

Click the **Create New Virtual Device** button to begin configuring a new *Android virtual device (AVD)*. Let's begin by choosing the type of device, as shown in Figure 4-20. Then click **Next**.

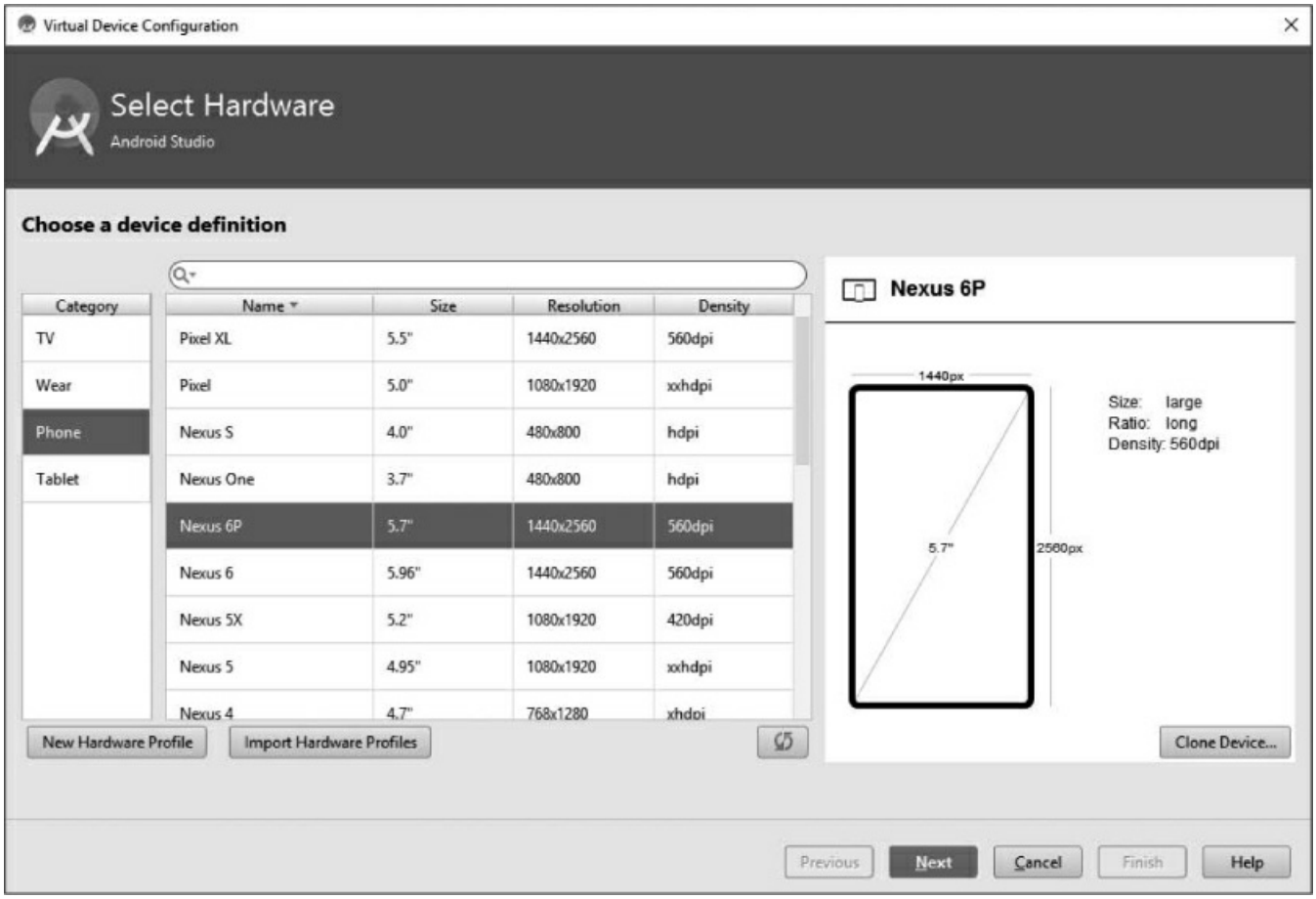


Figure 4-20: Choose a device to emulate.

We'll select a Nexus 6P to start with, but feel free to test out other devices. If you decide to develop your own app for the Google Play Store, you'll want to test it across a variety of screen sizes and device types, so you can try setting up

several different emulators. But let's start small for now with just one.

Next, we need to select the system image, usually either x86 or ARM. We'll be using an ARM image, so click the **Other Images** tab, as shown in Figure 4-21.

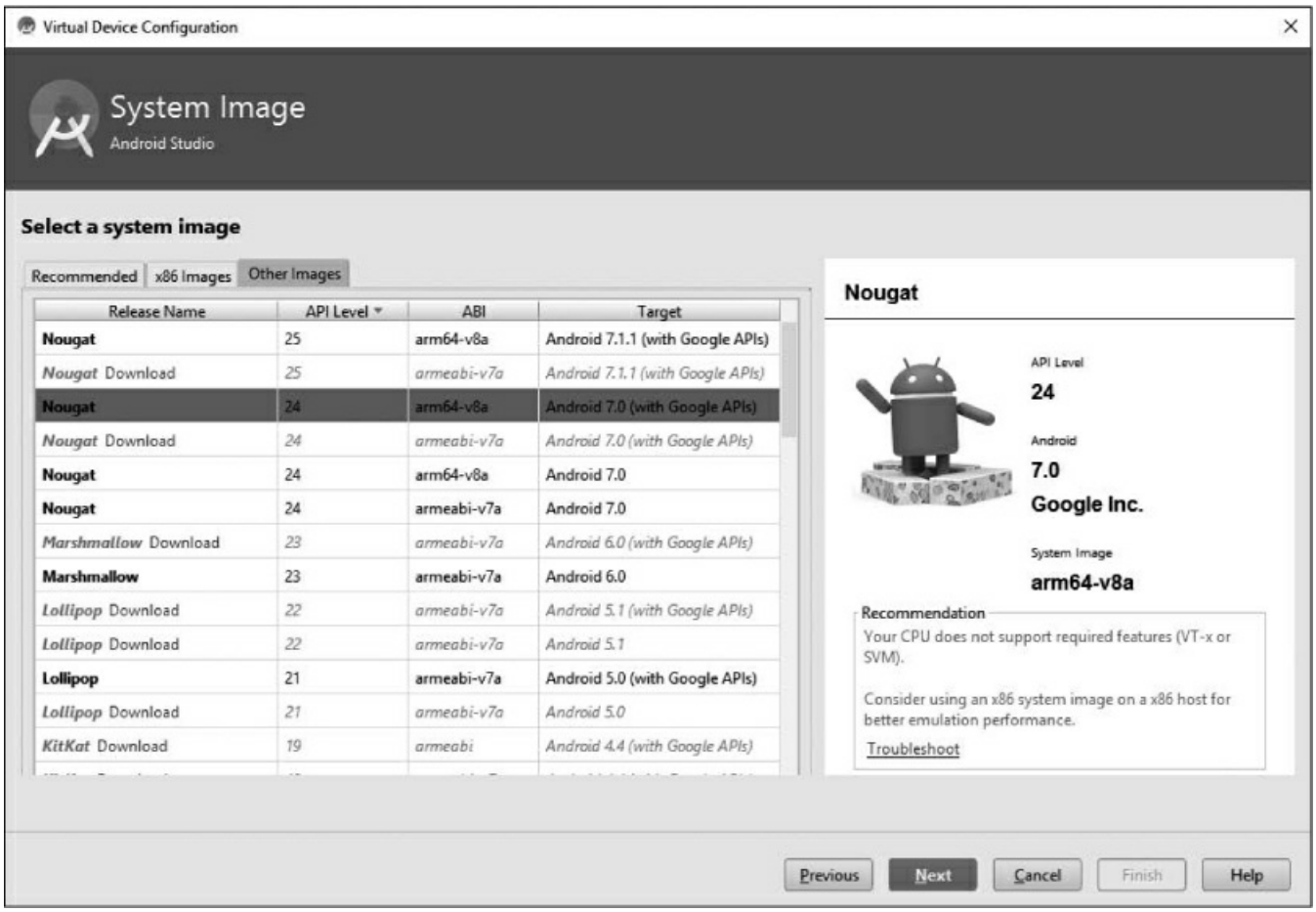


Figure 4-21: You can choose either an x86 or ARM emulator; ARM is slower but tends to work more consistently on different types of processors.

If the Android version, API level, or emulator you want is displayed in a gray font, click the **Download** link for the version you want.

Click **Next** when you've chosen the Android emulator you want. For now, let's use the Nougat API 24 ARM Android 7.0 version with Google APIs. You'll see a final screen asking you to verify your configuration, including any advanced settings changes you want to make. Let's change the name of the AVD, the device you're emulating, to *My Nexus 6P*, as shown in Figure 4-22.

If you have trouble getting your emulator to run on an older computer or on a computer with less than 8 GB of RAM, click **Show Advanced Settings**, scroll down to Memory and Storage, and change the RAM amount to **768 MB**.

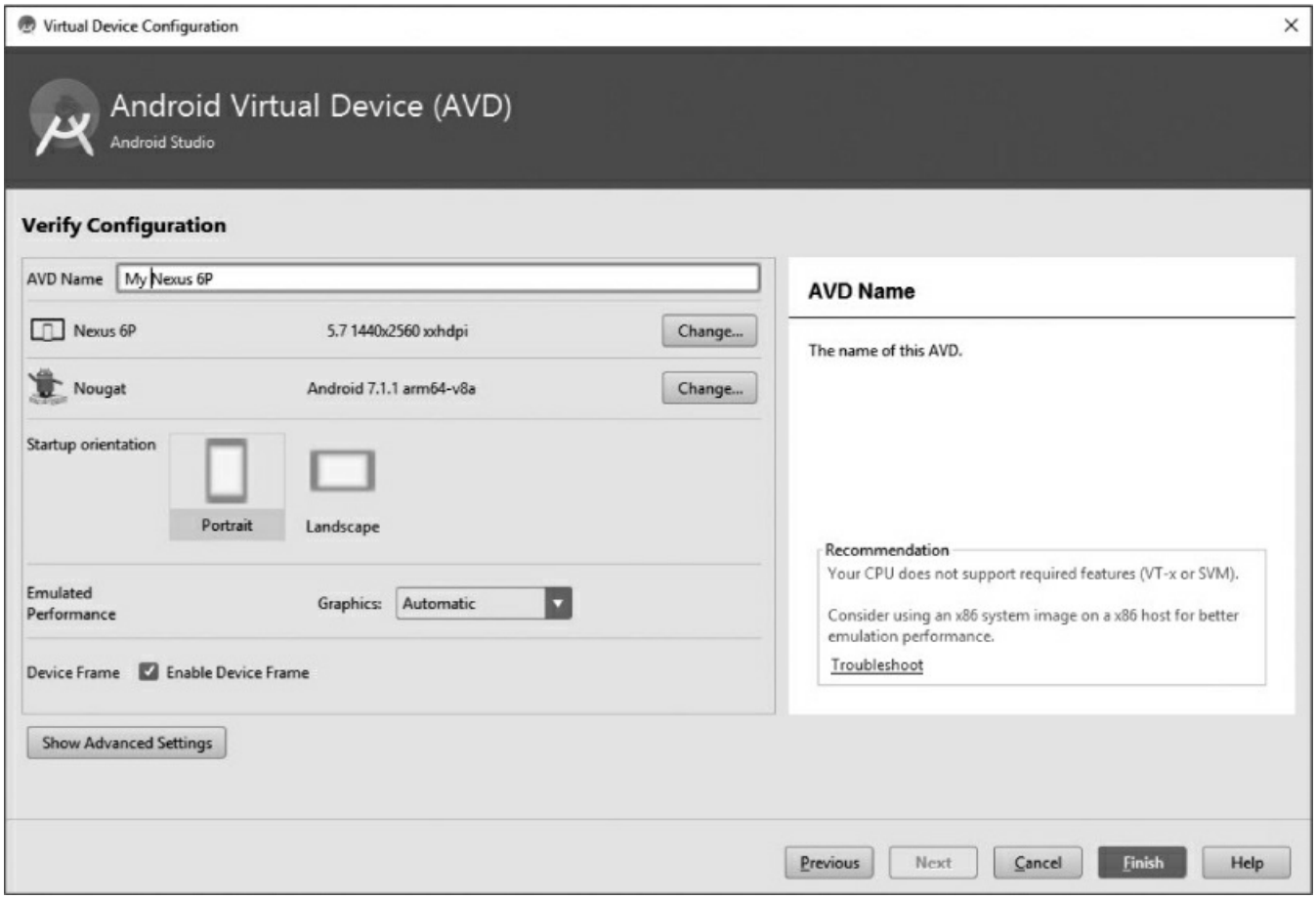


Figure 4-22: Name your new AVD.

Click **Finish**, and your new device will be created and saved to disk. Now it's time to start the emulator and try out your device. You should see the Select Deployment Target window again, but this time, it shows *My Nexus 6P* as an available emulator. Now click **OK**, as shown in Figure 4-23.

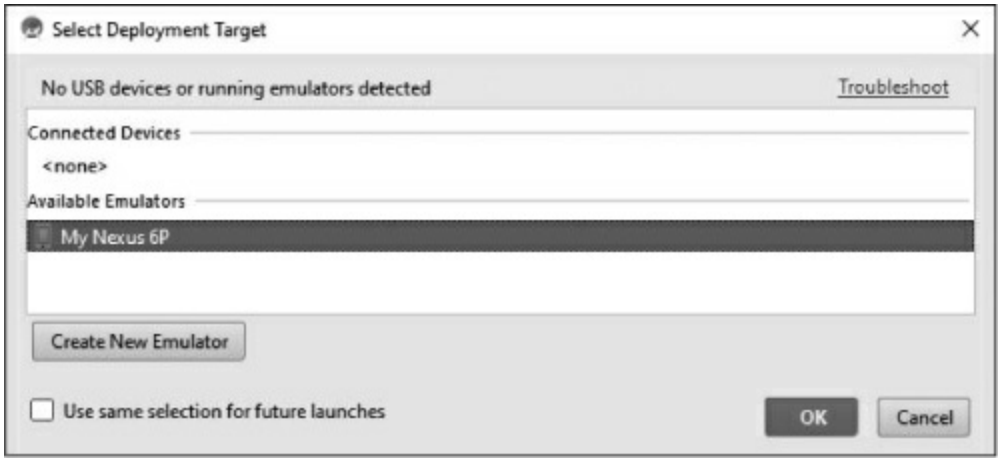


Figure 4-23: Choose the emulator you just created as your deployment target.

After a few moments, you'll see the message "Starting AVD," followed by an

emulator window that looks like an Android startup screen, as shown in Figure 4-24. This could take *several* minutes the first time you run the emulator. When the emulator starts up, you might see a lock screen.

Figure 4-24: The Android emulator booting up (left); the Android virtual device's home screen (right)

If your emulator shows a lock screen, click the lock at the bottom of the screen and drag it up to unlock the virtual device, similar to swiping the lock upward on an actual Android phone. You might see a welcome screen or two, but feel free to click out of those until you reach the home screen, which is shown in Figure 4-24.

Now, go back to Android Studio and press the run button again. This time, you'll see your AVD in the Select Deployment Target dialog (see Figure 4-25). Choose your device and click **OK**.

Figure 4-25: The emulator appears in the list of connected devices when running.

The project will build one more time, and then it will transfer the executable version of the app to the emulator. (You may be asked to update your Android SDK Tools the first time you run the emulator.) After a few minutes, you'll see your guessing game app running on the emulator!

We've coded the full app, so you should be able to play the game right away on your emulator. Take a look at Figure 4-26 to see what a round of the game might look like on the app.

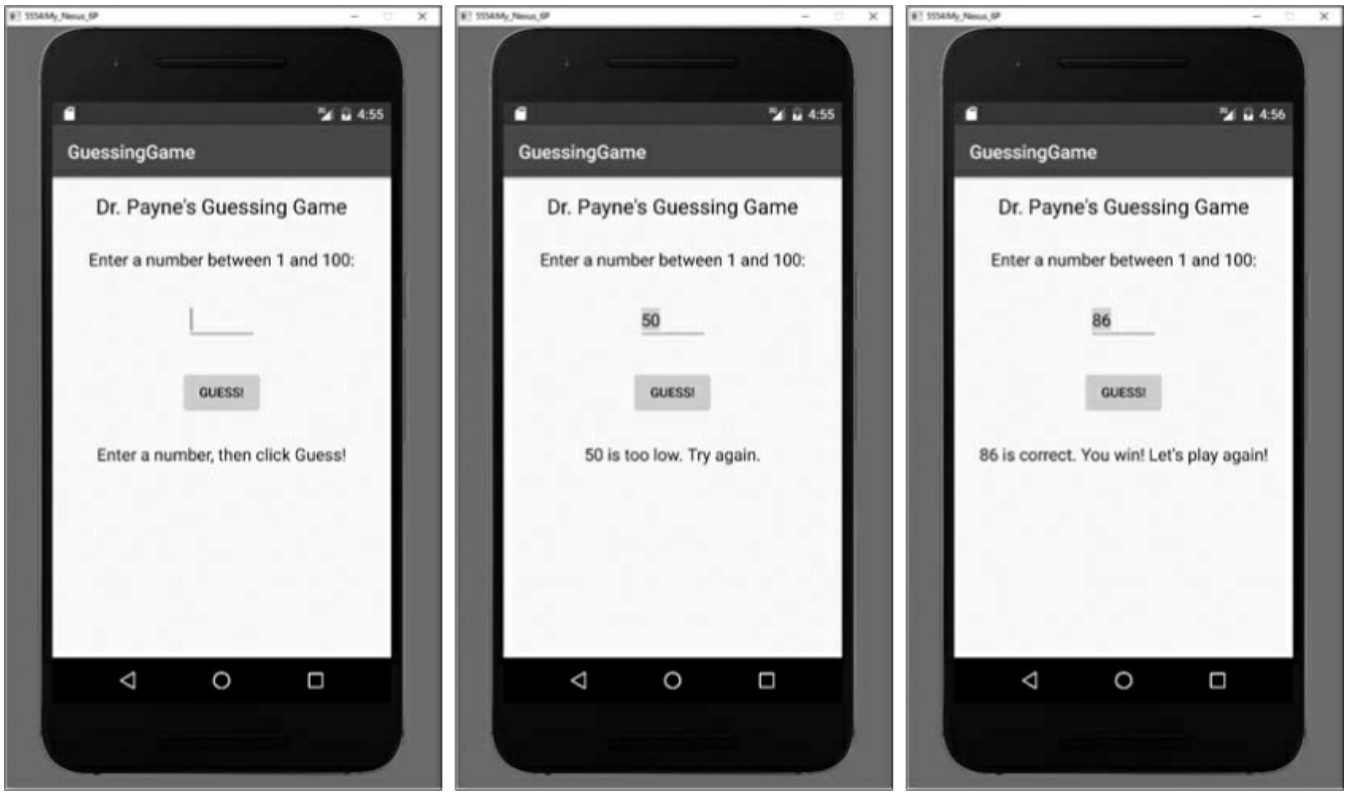


Figure 4-26: The guessing game app running on the emulator screen (left), after one guess (center), and after a winning guess (right).

It works just like the desktop app, but on an Android emulator. Use your keyboard to type a guess and your mouse to click the Guess! button. There are a couple more things we'll do to improve the user experience, just like we did with the GUI desktop app, but our Android mobile app is fully functional!

Before we improve the UX, we'll cover how to run the app on a real Android device. For now, leave your Android emulator open. You can minimize it when you're not using it, but leave it running in the background while programming to avoid the long startup time.

> **NOTE**
>
> *If you don't have an Android device, you can skip to "Improving the UX" on page 102.*

Running the App on a Real Android Device

Running the app on a real Android device can take some time and preparation, but if you have a USB cable to connect your device to your computer, you'll be up and running quickly.

Preparing Your Device

Before you can deploy your own apps to your Android device, you have to enable Developer mode on the device. You'll also need to change a couple of settings to allow you to develop and debug apps.

On your Android device, tap **Settings** and then scroll to the bottom of the Settings menu to find the About tablet, About phone, or About device section. At the bottom of the About screen, you'll find the Build number entry. Tap the build number at least seven times to enable the secret Developer mode. Developer mode allows you to test the apps you develop on your own device. Figure 4-27 shows the Settings menu (left) and the About menu (center) with Developer mode unlocked.

After you've unlocked Developer mode, tap the back arrow in the upper-left corner of the About screen and open **Settings ▸ Developer options**. Make sure the Developer options are turned **On**, as shown in Figure 4-27 (right). You

should also turn on USB debugging.

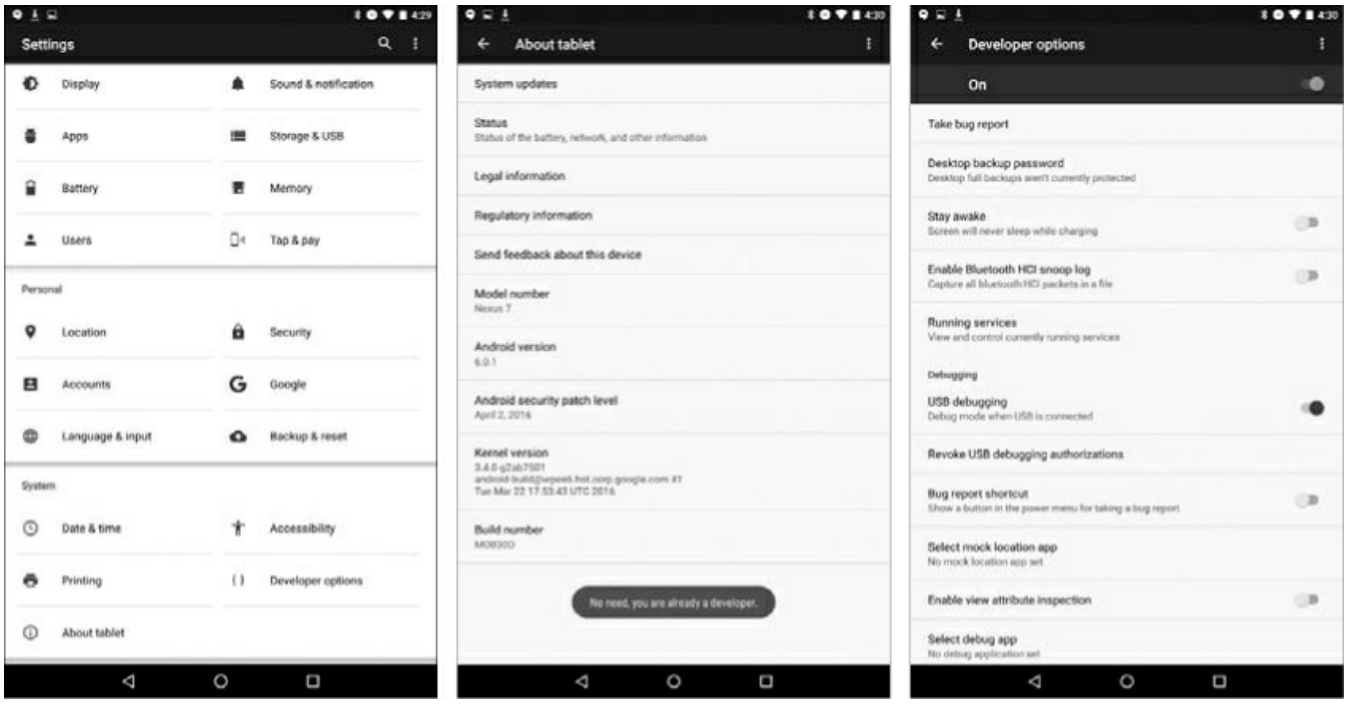


Figure 4-27: The Settings menu on an Android Nexus 7 tablet (left), the About menu (center), and the Developer options (right).

Now you're ready to connect an Android phone, tablet, or other device to your computer and run your app.

Connecting Your Device

To connect an Android device to your computer, you'll need a USB cable, preferably the one that came with your phone or tablet. It should be a micro-USB cable, and it is often supplied as part of the charger for your device. Note that not all charger cables are fully functional USB cables—if your cable doesn't work after you complete the following steps, try a different one.

Connect your device to your computer using the USB cable. After you plug in, the screen on your phone or tablet will show a window like the one in Figure 4-28 asking whether you want to allow USB debugging from the computer you just attached to. Tap **OK**. You can select the checkbox beside **Always allow from this computer** to prevent the pop-up next time you connect by USB to the same computer.

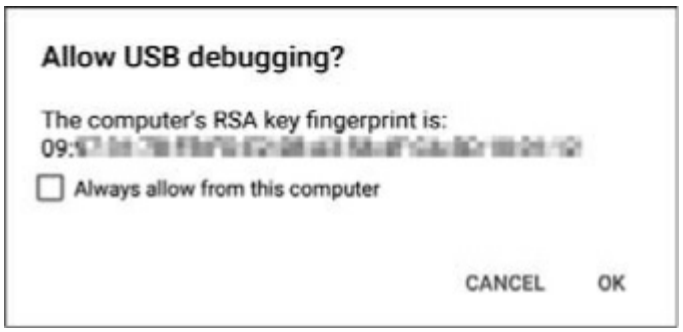


Figure 4-28: When you connect your Android device to your computer for the first time, it will ask whether you want to allow USB debugging.

Allowing USB debugging enables a couple of things. First, we'll be able to transfer the apps we program right to the Android device by transferring an *Android package file (APK)*. This is usually much faster than running apps on an emulator, and it's a better way to test how the apps will behave on a real device. Also, we'll be able to debug apps using the USB connection, meaning we'll get information back from the Android device to help us debug. This information will include errors and other log entries that we'll be able to read in a console-like application (called *logcat* in Android Studio). This is similar to the console output we used to find bugs in our command line and desktop GUI Java apps.

Now that you've connected a real Android device to your computer, let's see how to run the guessing game app on it!

Running the App on Your Device

Now we're ready to try the app on the Android device. In Android Studio, press the run button or go to **Run ▸ Run 'app'**. This time, the Select Deployment Target window will show *two* connected devices to choose from, as shown in Figure 4-29: your emulator (My Nexus 6P) and your real Android device.

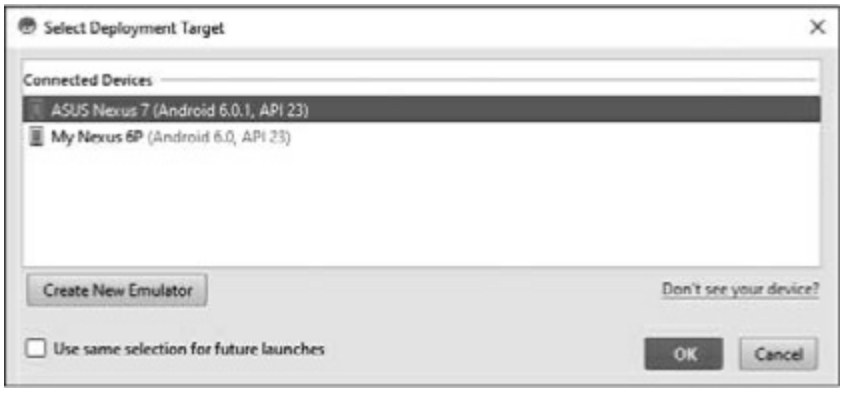


Figure 4-29: If you've unlocked Developer mode, enabled USB debugging, and connected your device

successfully, you should be able to select your Android device to run your app.

Choose your real Android device this time and click **OK**. Android Studio may take a moment to build the application, but once it transfers the APK file to your device, you'll see the guessing game app open right up on your tablet or phone. Play a round or two to verify that it works just as it did on the emulator (maybe even better, or at least faster). Very cool!

You may notice one important difference right away: the bottom part of the screen is covered by the numeric keypad, as shown in Figure 4-30.

When we placed the text field (`EditText`) for the user's guess, we chose a Number text field for our input so the numeric keypad would show up. But since the emulator runs on a computer, which has its own keyboard, you probably didn't see this when you ran your app earlier.

Now that we're able to run Android apps on an Android phone, tablet, or other device, let's make a few final improvements to the user experience for the guessing game app.

Figure 4-30: The app runs on your own Android device and shows the numeric keypad by default.

Improving the UX

The guessing game app works beautifully on both the Android emulator and on real Android devices, but there are a few UX items we can improve upon. First, we'll center the user's guess in the Number text field. Second, we'll learn how to treat the ENTER key just like the Guess! button to make the app more intuitive to the user.

Centering the User's Guess in the Text Field

In Android Studio, open the *content_main.xml* layout file again. Select the text field, called `txtGuess`, by clicking it in the design preview. Under the Properties pane, find **textAlignment**. The `textAlignment` property in Android is similar to the `horizontalAlignment` property in the desktop GUI—it allows us to change the alignment of the text the user enters.

Click the center text alignment option, the icon that looks like centered text. If you want to test it in the design preview without running the app, scroll to find the `text` property and enter a value of **`50`**. The number should appear centered in the text field in the design preview pane. Be sure to delete the test value if you don't want it to show up in the actual app.

Adding a Listener for the Enter Key

Let's set up an event listener to handle the ENTER key the same way we handled a click on the Guess! button, with a call to `checkGuess()`.

Go back to the *MainActivity.java* source code file, scroll down to the `onCreate()` method, and find where we added an `onClickListener` to the button `btnGuess`. (You may need to *expand* the event listener code by clicking the + in the left margin beside `btnGuess.setOnClickListener()` to make it look like the code shown here.) Right below that, we can add an event listener to the text field `txtGuess` to handle action events such as pressing the ENTER key. The following code adds a listener to `txtGuess`:

```
btnGuess.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        checkGuess();
```

```
    }
});
txtGuess.setOnEditorActionListener(new TextView.OnEditorActionListener() {
    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
        checkGuess();
      ❶ return true;
    }
});
```

The *editor action* we're listening for is when the user presses the ENTER key while typing in the text field. When that event happens, we want to check the guess the user entered.

We changed the `return` statement to return a value of `true` ❶ because we want to keep the keyboard on the screen so the user can enter their next guess. The `return` statement tells the text field whether the event handler code we've provided completed the event. By returning the value `true`, we've told Android that we've done everything that needs to be done to check the user's guess. If we returned `false` here instead, Android would finish processing the ENTER key by removing the number keypad from the screen—as it does by default when you finish typing in a form on a web page. We don't want the keypad to disappear after every guess, so we return `true`.

Adding One More Finishing Touch

Run the app a few times to test it, and you'll see the ENTER key allows you to enter and check guesses quickly and efficiently. You'll also see your guesses centered nicely in the text field. However, when you win, depending on the API version, screen size, or density of your device, you may see an alignment problem like the one shown in Figure 4-31.

The problem is that the `TextView` named `lblOutput` resizes to show the longer "You win! Let's play again!" text, and some older APIs may not center this correctly. You can prevent this by expanding `lblOutput` to the full width of the screen in the design preview and by changing the `textAlignment` property to `center`. Now you have a complete guessing game app to play on your device and to share with friends! The final version 1.0 of the app is shown running on a Nexus 7, with Font settings at Large, in Figure 4-32.

7:40
GuessingGame
Dr. Payne's Guessing Game
Enter a number between 1 and 100:
78
GUESS!
78 is correct. You win! Let's play again!
- + . 1 2 3
* / , 4 5 6
( ) = 7 8 9
* 0 #

Figure 4-31: If you notice misaligned text, you may need to expand and center the TextView `lblOutput`.

Figure 4-32: Your finished version 1.0 of the mobile app, with all the user experience improvements in place

Wow! You've gone from a simple, text-based, command line number-guessing program, to a sophisticated GUI desktop game, to a fully featured mobile app that runs on real Android devices. Are you starting to see the power and flexibility of Java? And in Chapter 5, we'll make the app even more professional with a settings menu and the ability to store user preferences!

What You Learned

You've seen that much of the Java code used in a text-based and GUI desktop program can build the foundation of a modern mobile app, thanks to Java's reusability across multiple platforms. We have also picked up several new mobile-specific skills in this chapter:

- Starting a new Android Studio app project

- Building the GUI layout in design view in Android Studio, including changing the properties of various elements in the Properties pane
- Naming GUI components in a layout for easy use in Java code
- Connecting the Android GUI layout elements to your Java code
- Adding your own methods, such as `checkGuess()` and `newGame()`, to an Android app
- Reusing Java code in an Android app
- Handling events in Android, including button clicks and keyboard/editor actions such as pressing the ENTER key
- Testing apps by running them on an Android virtual device using the Android emulator
- Running an app on an Android phone, tablet, or other device by enabling Developer mode and USB debugging
- Improving the user experience by changing widget properties and adding user-friendly finishing touches

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned and to expand your programming skills by trying something new. If you get stuck, visit *https://www.nostarch.com/learnjava/* to download sample solutions.

#1: "Toast"-ing to the Number of Tries

In the programming challenges for

Chapter 3, you changed the winning message to tell the user how many tries they took to win:

```
62 is correct! You win after 7 tries!
```

Let's do something similar for the Android version, with a twist—or, should I say, a *Toast*? A Toast is an Android widget for creating pop-up message windows. Toast pop-ups are a convenient way to show a quick notice, such as an error or a note telling the user they've won the game. See the Toast message near the

bottom of the screen in Figure 4-33.

Figure 4-33: A Toast message pops up on the screen to alert the user to important information.

You can create a Toast message popup with the `Toast.makeText()` method. Here's a sample to help you add a Toast pop-up to the `else` statement for winning the game:

```
else {
    message = guess +
            " is correct. You win after " + numberOfTries + " tries!";
    Toast.makeText(MainActivity.this, message,
            Toast.LENGTH_LONG).show();
    newGame();
}
```

The Toast message in this code will pop up over the `MainActivity` (our guessing game), displaying the `String message` for a few seconds thanks to `Toast.LENGTH_LONG`. There's also a `Toast.LENGTH_SHORT`, but it pops up and disappears so quickly that it can be hard to read. The `.show()` method at the end

does the important part, showing the Toast on the screen.

As with the GUI desktop version, to accomplish this task, you'll need to create a new variable at the top of your class (like `int numberOfTries = 0;`), add to the number of tries every time the `checkGuess()` method runs successfully, and change the output `message` to display the number of tries to the user when they win, both in a Toast pop-up and in the `lblOutput` field. After counting the number of tries, add the features you learned in Chapter 3 for running out of tries or losing. Give the user seven tries and tell them how many tries are left after each guess.

#2: Adding Visual Appeal

Explore some of the more aesthetic properties of the GUI components in Android Studio, including background colors, foreground colors, fonts, `textSize`, and more. Move components in your guessing game app around to make the interface more visually appealing. For an example of one way you could customize your game, see Figure 4-34.

You can even add a custom background image (add an image file to the *app* ▸ *res* ▸ *drawable* folder and then select it as the `background` property for your layout or for a button or other component). Play with various settings—you can always undo by pressing CTRL-Z!

Figure 4-34: Changing the colors, fonts, text size, and arrangement of the components in your app can make it stand out!

#3: Creating a MadLibs Mobile App

Look back at the *MadLibGUI.java* program you built in Chapter 3 (Programming Challenge #3 on page 74). Now create a mobile version of your MadLibs app that prompts the user for several words in a GUI with labels and text fields, such as `txtBigAnimal`, `txtPastTenseVerb`, `txtSmallContainer`, and `txtFood`. Add a button that the user can press to generate their own MadLibs-style story. It may be a nice touch to include some default or initial text in each text field, just to give the user an idea of how the program works.

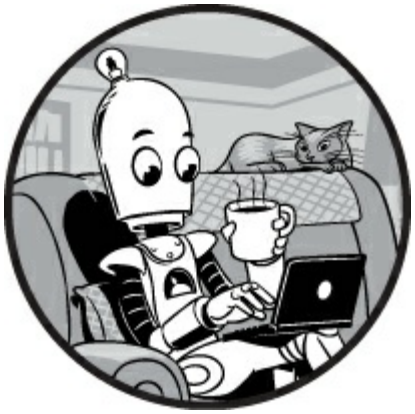
When the user clicks the button, the program should display the completed MadLibs story in a `TextView` or in an `EditText` widget—the `EditText` makes it possible to copy and paste the user's funny story if they wish to share it. Play with the colors, fonts, and layout until you're happy with the result and then share your creation with a friend!

HINT

To make a TextView or EditText show multiple lines, use the escape character sequence \n to insert a new line in the String message you display using the setText() method, like so: "Once upon a time... \n There was a buffalo princess \n who lived in a soup can." This quote will span three lines, separated by new lines wherever the \n escape sequence appears.

5

POLISHING YOUR APP BY ADDING MENUS AND PREFERENCES



You've written a fun Android app, but it's still missing a few things. You haven't learned how to build a settings or options menu in Android or how to save high scores, game stats, and other information. In this chapter, we'll add an options menu to our Hi-Lo guessing game app and the capability to store information.

Adding an Options Menu in Android

Most apps and games have options or settings that the user can access through a menu. For the Hi-Lo guessing game, we might want to allow the user to change the game's difficulty level, start over, see their game stats, or view an About screen, so we'll create a menu that can perform all of these actions.

There are four general steps to adding an options menu in Android:

1. Edit the app's default XML menu file to create items that will serve as the options the user can select.
2. Modify the app's activity file to display the menu and options that we created in the previous step.
3. Create an event handler to determine when the user selects an option.

4. Write the code to execute for each option when the user selects it.

Adding the options menu not only will make our app look and feel more professional but will also give the user greater control over their gaming experience. My sons loved being able to change the range of the guessing game from 1 to 10, then 1 to 100, then 1 to 1,000, but when we added the Game Stats option that displays the number of games won, I almost couldn't get my device back from them—they wanted to keep running the number higher and higher! I hope you'll find these extra features as fun (and maybe as addictive) as they did.

Adding Items to the Menu's XML File

Open your Hi-Lo guessing game project in Android Studio and, in the Project Explorer pane, change the view at the top left to **Android**. Then, open the default menu file by expanding **app ▸ res ▸ menu** and double-clicking *menu_main.xml*.

Change the XML code in your *menu_main.xml* file to match the following listing:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_settings"
        android:title="Settings" />
    <item
        android:id="@+id/action_newgame"
        android:title="New Game" />
    <item
        android:id="@+id/action_gamestats"
        android:title="Game Stats" />
    <item
        android:id="@+id/action_about"
        android:title="About" />
</menu>
```

The `<menu>` tag creates a menu resource using the XML namespace for Android XML documents, identified by the uniform resource identifier (URI) *http://schemas.android.com/apk/res/android*. We can use XML to store or display everything from web pages to databases by connecting XML tags to those elements. The `xmlns` (XML namespace) attribute in this code selects the main Android namespace so that the tags in this XML file will refer to common elements in an Android application. So, the `<menu>` tag refers to an Android menu, and each `<item>` tag describes an item or entry in that menu with

attributes. This menu will have the four options Settings, New Game, Game Stats, and About, so we add four `<item>` tags. We assign those names to each item's `title` attribute, which determines the text displayed for each option when the user opens the menu. We'll use the `id` attribute later in our code to determine which option the user has selected.

Save the *menu_main.xml* file. Now it's time to display our options menu in the Hi-Lo guessing game app.

Displaying the Options Menu

We've set up the menu, but in order to display it, we need to add some Java code to our app's *MainActivity.java* file. Open *MainActivity.java* in the Project Explorer pane under **app ▸ java ▸ com.*yourdomain*.GuessingGame**.

Near the middle or bottom of the `MainActivity` class, you should find a method called `onCreateOptionsMenu()`. Modify it to match the code snippet below. (If your code doesn't have the `onCreateOptionsMenu()` method, add the following code after the closing brace of `onCreate()` but before the final closing brace for `MainActivity`.)

```
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu_main, menu);
        return true;
    }
} // Final closing brace of the MainActivity.java file
```

The `onCreateOptionsMenu()` method does exactly what its name implies—it tells Android what to do when creating the options menu for our app. In this case, we tell Android we want to expand the *menu_main.xml* file to serve as our options menu. The *menu_main.xml* file isn't a menu yet, so we need to convert it into a menu using a class called a `MenuInflator`. We'll create a `MenuInflator` instance, which we call `inflater`, by using the `getMenuInflator()` method. Once we have `inflater`, we call the `inflate()` method and pass it the XML file (`R.menu.menu_main`) and the menu we want the XML file's items to inflate into (`menu`). You may need to press ALT-ENTER (on macOS, OPTION-ENTER) to correct any missing `import` statements as you add the code to your file.

After making this change, save and run the app. Android lets you know there's an options menu available by displaying the three dots in the action bar of your app (see Figure 5-1, top). Clicking the dots will display the options menu (see

Figure 5-1, bottom).

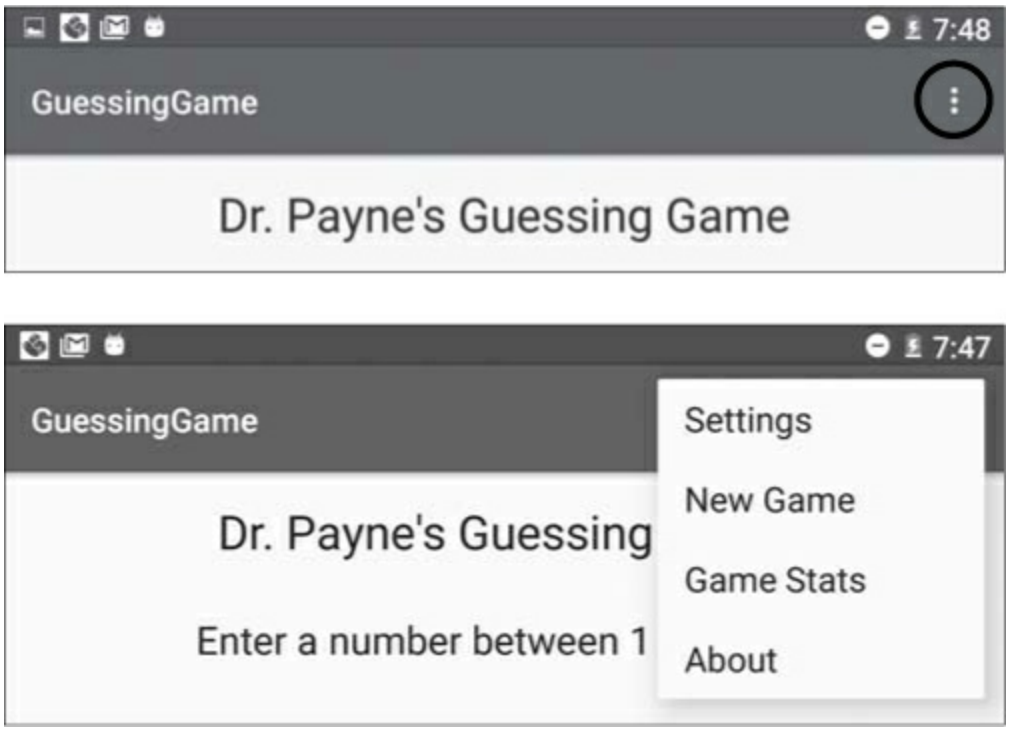


Figure 5-1: The options menu shows up as three dots in the app's action bar (top). Clicking the three dots will expand the options menu (bottom).

You'll notice that clicking the options doesn't do anything yet because we haven't added the code to respond to user selections. We'll do that next.

Responding to User Selections

When the user chooses an option from the menu, we want our app to perform the requested action. In order for our app to do that, we need to add an event handler that keeps track of which option was selected. We'll use the `id` attribute associated with each item to tell selections apart.

In the *MainActivity.java* file, find and modify the `onOptionsItemSelected()` event handler method. Alternatively, you can add it right below the method `onCreateOptionsMenu()` that we modified in the previous section, but before the closing brace on the final line of the file.

```
    public boolean onCreateOptionsMenu(Menu menu) {
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.menu_main, menu);
        return true;
    }
    public boolean onOptionsItemSelected(MenuItem item) {
```

```
        switch (item.getItemId()) {
            case R.id.action_settings:
                return true;
            case R.id.action_newgame:
                newGame();
                return true;
            case R.id.action_gamestats:
                return true;
            case R.id.action_about:
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
}
```

In this code, we use a `switch` statement to determine which of the options the user selected on the menu. A `switch` statement is another way of testing multiple conditions, similar to a long chain of `if-else` statements. Rather than chaining four `if-else` statements to test for each possible menu item selection, though, we can use a single `switch` statement, with the variable we're testing inside parentheses after the `switch` keyword. In this example, we're checking the `id` of the menu item the user selected, so we use `switch (item.getItemId())`. Then, inside the braces for the `switch` statement, we list the values we want to test as `case` statements (e.g., `case R.id.action_settings`), each of which is followed by a colon (:), the code to run for that selection, and either a `break` or `return` statement. This event handler returns a Boolean value, so we've used `return` statements instead of `break` statements in each `case` block. If we didn't have `return` statements here, we would need to use a `break` command as the final statement of each case.

Each `case` statement in this code tests for one of the `id` values of the items we entered in the *menu_main.xml* file, and the code for each value will execute in response to the user's selection. Right now, we have code only for the `action_newgame` case, which starts a new game using the `newGame()` method. The other cases require writing a bit more code, so we'll define those one at a time.

Creating an Alert Dialog Pop-up for the About Screen

For the About menu option, we'll pop up a dialog window similar to other applications you've likely seen. To do this, we'll use an *alert dialog*, a flexible pop-up used to inform the user about something or to prompt them for a response. This kind of pop-up is more adaptable than the Toast pop-up we used

in Chapter 4 (Programming Challenge #1 on page 106), because the `AlertDialog` class allows us to customize the properties of our dialog through the `Builder` subclass. In this case, we'll use an alert dialog to respond to the user selecting the About option with a message telling them who created the awesome Hi-Lo guessing game they've been playing.

Add the following code to the `case` statement for the `action_about` item selection:

```
case R.id.action_about:
  ❶ AlertDialog aboutDialog = new
AlertDialog.Builder(MainActivity.this).create();
  ❷ aboutDialog.setTitle("About Guessing Game");
  ❸ aboutDialog.setMessage("(c)2018 Your Name.");
  ❹ aboutDialog.setButton(AlertDialog.BUTTON_NEUTRAL, "OK",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                  ❺ dialog.dismiss();
                }
            });
  ❻ aboutDialog.show();     return true;
```

We use the `AlertDialog.Builder` class ❶ to create a customized popup window. The code at ❷ sets the title of the pop-up window to `"About Guessing Game"`, and the line at ❸ displays a simple message with copyright information and your name (though you can write any text you'd like here). The `setButton()` method ❹ adds a button to the pop-up with the text `"OK"`, and the subsequent `onClick()` event listener closes the pop-up whenever the user clicks the OK button by calling the `dismiss()` method ❺. Finally, the customized pop-up is shown with the `show()` command ❻.

Use ALT-ENTER (or OPTION-ENTER) to import the `AlertDialog` class. Then save your updated code and run the new version of the app. When you click the options menu and select the About option, you should see a pop-up window like the one in Figure 5-2.

Figure 5-2: A custom alert dialog pop-up

The Hi-Lo guessing game app is beginning to feel even more like a professional Android app! Now let's knock it out of the park by allowing the user to choose the difficulty level of the game and to keep track of how many games they've won.

Changing the Guessing Range

Letting the user choose the guessing range—say, from 1 to 10, 1 to 100, or 1 to 1,000—would be a huge enhancement. Now that you understand option menus and alert dialogs, let's map out how we might upgrade the game to allow the user to change the range setting.

First, we'll need to add a variable for the range so that instead of using a hardcoded value of `100`, we'll use the user's chosen range. Second, we'll need to modify the app's behavior in a couple of ways. We'll change the `newGame()` method to use the new range variable. We'll also make the `TextView` that currently reads, `"Enter a number between 1 and 100:"` display a different prompt depending on the range selected. Finally, we'll need to give the user a method of choosing the range. We'll do this by building another custom alert dialog with the three range selections: 1 to 10, 1 to 100, and 1 to 1,000.

Adding a Variable for the Range

First, we'll replace the hardcoded value of `100` we use in the random number calculation with a variable. At the top of the `MainActivity` class, add a variable for the `range` and set it to a default value of `100`:

```
public class MainActivity extends AppCompatActivity {
```

```
    private EditText txtGuess;
    private Button btnGuess;
    private TextView lblOutput;
    private int theNumber;
    private int range = 100;
```

While we're adding variables, let's also add a second `TextView` for the label that reads, `"Enter a number between 1 and 100:"`. When the user selects a range other than 1 to 100, this label will no longer be correct, so we need a variable to store the appropriate text to display. We'll create the variable called `lblRange` to do this:

```
    private int range = 100;
    private TextView lblRange;
```

To wire up the GUI to the `lblRange` variable, add the following line of code to the
`onCreate()` method:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    txtGuess = (EditText) findViewById(R.id.txtGuess);
    btnGuess = (Button) findViewById(R.id.btnGuess);
    lblOutput = (TextView) findViewById(R.id.lblOutput);
    lblRange = (TextView) findViewById(R.id.textView2);
```

If you get an error, check the name of your prompt's `TextView` in the design view: open **app ▸ res ▸ layout ▸ *content_main.xml*** and click the label reading `"Enter a number between 1 and 100:"`. Change the label's `id` property to `textView2`.

With the `range` and `lblRange` variables set up, it's time to modify the behavior of the app to use these variables instead of hardcoded values.

Using the range Variable

First, let's modify the `newGame()` method to use the `range` variable. Let's also add the code needed to change the prompt to let the user know the correct range to guess from:

```
public void newGame() {
    theNumber = (int)(Math.random() * range + 1);
    lblRange.setText("Enter a number between 1 and " + range + ".");
    txtGuess.setText("" + range/2);
    txtGuess.requestFocus();
    txtGuess.selectAll();
```

```
}
```

In addition to using `range` to set the random number correctly, we've changed the `lblRange` prompt to use the `range` variable as well. The last three lines are a finishing touch—I've taken the liberty of entering a default starting value in the `txtGuess` text box, one half of the `range`. So, if the user is guessing between 1 and 10, the guessing text field will show 5 as a default first guess; if the range is 1,000, the text field will recommend 500 for the first guess.

One last range-related change is found in the `checkGuess()` method. We added a `try-catch` statement to handle bad user input, and in the `catch` statement, we tell the user to enter a valid whole number in the range from 1 to 100. Let's change just the `catch` statement to reflect the user's selected range:

```
public void checkGuess() {
--snip--
        }
    } catch (Exception e) {
       message = "Enter a whole number between 1 and " + range + ".";
     } finally {
--snip--
    }
}
```

Now both `TextView` labels will display the user's selected range correctly. It's time to build the alert dialog to allow the user to select the difficulty level of their game.

Building the Dialog to Allow the User to Select the Range

The range settings dialog should display all the range options (1 to 10, 1 to 100, and 1 to 1,000) to the user whenever they select the Settings option in the menu. To display the list of options, we'll build another customized alert dialog, but this one will display a list view of the three range options.

First, scroll back down to your `onOptionsItemSelected()` method and add the following code inside the `case` statement for `action_settings`:

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_settings:
             final CharSequence[] items = {"1 to 10", "1 to 100", "1 to 1000"};
             AlertDialog.Builder builder = new AlertDialog.Builder(this);
             builder.setTitle("Select the Range:");
             builder.setItems(items, null);
```

```
            AlertDialog alert = builder.create();
            alert.show();
            return true;
```

These six lines of code will display an alert dialog with a list view of the three options for the guessing range, but we need to add just a bit more code to handle the user's selection. The `builder.setItems()` method will accept both a list of items and an event listener to handle the user's choice from the list.

If the user selects the first option, we need to change the value of the `range` variable to `10`, and likewise for the second and third options to `100` and `1000`, respectively. The code for the event listener goes inside the `builder.setItems()` statement:

```
case R.id.action_settings:
   final CharSequence[] items = {"1 to 10", "1 to 100", "1 to 1000"};
   AlertDialog.Builder builder = new AlertDialog.Builder(this);
   builder.setTitle("Select the Range:");
   builder.setItems(items, new DialogInterface.OnClickListener() {
       public void onClick(DialogInterface dialog, int item) {
          switch(item) {
             case 0:
                  range = 10;
                  newGame();
                  break;
             case 1:
                  range = 100;
                  newGame();
                  break;
             case 2:
                  range = 1000;
                  newGame();
                  break;
          }
          dialog.dismiss();
       }
});
AlertDialog alert = builder.create();
alert.show();
return true;
```

Notice that after we set the new range for each item selection, we call `newGame()` to create a new random number in that range and to change the prompt on the user's screen to reflect the new range.

Save the file after making these changes and run the game to test the new options. Change the range from 1 to 10 and guess a few rounds, then go back to 1 to 100, and if you're brave, go for 1 to 1,000.

Close the app and open it again, however, and you'll notice that the game doesn't remember your preferred range when it's run a second time. The app also doesn't remember how awesome you are at guessing the correct number. If only there were a way to have the app remember your preferred range and the number of games you've won. . . .

Storing User Preferences and Game Stats

The key to remembering user preferences and game statistics from one session to another is the ability to save *persistent information* to your Android device. Persistent information is any data that remains on the device after the app is closed. In the Hi-Lo guessing game, we want to store the user's preferred difficulty level and the number of games they've won as persistent information.

There are three ways to save persistent data to your Android device: storing shared preferences, saving files, and saving data in a database. *Shared preferences* are a type of object that stores a relatively short list of settings that your app needs to save for the next time you use the app. They're called *shared* preferences because you can share the settings across several activities or screens in your app, such as an options menu and the main game screen in the guessing game. Saving a file to the device is useful when you need to store a large amount of data, such as a text document, and databases are necessary for apps such as an address book or contacts list. But for the guessing game, we just need to store a few numbers, so we'll use shared preferences.

Storing and Retrieving the User's Preferred Range

Shared preferences are stored as sets of *key/value pairs* where each value has an associated key that is used to retrieve it. For example, you could have a pair like `"range"` and `"100"`, where `"range"` is the *key* and `"100"` is the *value* we're storing under that key. Let's write a method to store the user's preferred range to shared preferences.

Near the bottom of your *MainActivity.java* file, add the following method after the `onOptionsItemSelected()` method and just before the closing brace:

```
        default:
            return super.onOptionsItemSelected(item);
    }
```

```
  }
  public void storeRange(int newRange) {
       SharedPreferences preferences =
           PreferenceManager.getDefaultSharedPreferences(this);
       SharedPreferences.Editor editor = preferences.edit();
       editor.putInt("range", newRange);
       editor.apply();
  }
}
```

There is already a default shared preferences object for every app you make, which you can access by creating a `SharedPreferences` object to connect to it. To do this, access the default object by calling `getDefaultSharedPreferences()` on the `PreferenceManager` object that creates and maintains lists of shared preferences. Remember to import as you go, or press ALT-ENTER (or OPTION-ENTER).

To write to shared preferences, we must use an `Editor` object, which allows us to edit individual shared preference values. To store a specific key/value pair, we use a `put` method, such as `putString` to store a string value, `putInt` to store an integer, `putFloat` to store a floating-point decimal value, `putBoolean` to store a true/false value, and so on. Every time the user selects a new range, we'll pass the `range` variable to the `storeRange()` method as `newRange`. In order to store `newRange` under the `"range"` key, we use editor .putInt("range", newRange); to store the user's new range value (10, 100, or 1,000) under the shared preferences key name `"range"`. The `apply()` method tells Android you're finished modifying the shared preferences values, and it can apply the changes.

Now that we can store the range to shared preferences, we need to add the `storeRange()` function to each `case` the user can select in the event listener in the `onOptionsItemSelected()` method:

```
public boolean onOptionsItemSelected(MenuItem item) {
     switch (item.getItemId()) {
        case R.id.action_settings:
            final CharSequence[] items = {"1 to 10", "1 to 100", "1 to 1000"};
            AlertDialog.Builder builder = new AlertDialog.Builder(this);
            builder.setTitle("Select the Range:");
            builder.setItems(items, new DialogInterface.OnClickListener() {
               public void onClick(DialogInterface dialog, int item) {
                  switch(item) {
                       case 0:
                         range = 10;
                         storeRange(10);
                         newGame();
                         break;
                       case 1:
                         range = 100;
```

```
                storeRange(100);
                newGame();
                break;
            case 2:
                range = 1000;
                storeRange(1000);
                newGame();
                break;
            }
            dialog.dismiss();
        }
    });
    AlertDialog alert = builder.create();
    alert.show();
    return true;
```

Finally, we need to retrieve the range when the game loads so that the user's last selected range will be the range the game uses the next time it runs. Scroll up to the `onCreate()` method and add the following two lines to retrieve the range from shared preferences:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    txtGuess = (EditText) findViewById(R.id.txtGuess);
    btnGuess = (Button) findViewById(R.id.btnGuess);
    lblOutput = (TextView) findViewById(R.id.lblOutput);
    lblRange = (TextView) findViewById(R.id.textView2);
    SharedPreferences preferences =
        PreferenceManager.getDefaultSharedPreferences(this);
    range = preferences.getInt("range", 100);
    newGame();
```

Notice that we've retrieved the shared preferences *before* we call the `newGame()` method to ensure that the user gets the range they last used whenever the app starts over. The `getInt()` method looks for the value stored in the key `"range"`, but if it doesn't find a value, the second parameter tells it to default to `100`. We do this so that `range` has a value the first time the user runs the app.

Save the file, build, and run it. This time, choose a different range and then close the app entirely. The same range will be waiting for you the next time you start the app!

Storing the Number of Games Won

High scores, leaderboards, winning streaks—anything that records our accomplishments tends to make us want to try harder, play longer, and beat the

record. One finishing touch we'll add to the game is the ability to track the number of games won. Once again, we can easily store these stats as shared preferences.

When the user wins a round of the game by guessing the correct number, we can use a shared preference to retrieve the number of games they've won, add 1 to it, and store the new value. Add this code to your `checkGuess()` method, putting it inside the `else` statement for a winning guess:

```
public void checkGuess() {
     String guessText = txtGuess.getText().toString();
     String message = "";
     try {
       int guess = Integer.parseInt(guessText);
       if (guess < theNumber)
         message = guess + " is too low. Try again.";
       else if (guess > theNumber)
         message = guess + " is too high. Try again.";
       else {
          message = guess +
                " is correct. You win! Let's play again!";
       ❶ SharedPreferences preferences =
              PreferenceManager.getDefaultSharedPreferences(this);
       ❷ int gamesWon = preferences.getInt("gamesWon", 0) + 1;
       ❸ SharedPreferences.Editor editor = preferences.edit();
       ❹ editor.putInt("gamesWon", gamesWon);
       ❺ editor.apply();
         newGame();
    }
```

Here, we've accessed the default `SharedPreferences` at ❶, and at ❷, we've retrieved the value stored under the key name `"gamesWon"` (with a default of `0` if this is the first time the user has won) and added `1` to account for this win. At ❸, we create an editor to write a new value to shared preferences. At ❹, we put the integer value `gamesWon` into shared preferences under the corresponding key name for later use, and at ❺, we tell Android to write the changes to the device.

That takes care of storing the number of games won, but how about displaying those stats to the user? To do that, we'll need to add code for the `action_gamestats case` in the `onOptionsItemSelected()` method, as follows:

```
case R.id.action_gamestats:
  ❶ SharedPreferences preferences =
       PreferenceManager.getDefaultSharedPreferences(this);
  ❷ int gamesWon = preferences.getInt("gamesWon", 0);
```

```
   ❸ AlertDialog statDialog = new
AlertDialog.Builder(MainActivity.this).create();
     statDialog.setTitle("Guessing Game Stats");
   ❹ statDialog.setMessage("You have won "+gamesWon+" games. Way to go!");
     statDialog.setButton(AlertDialog.BUTTON_NEUTRAL, "OK",
          new DialogInterface.OnClickListener() {
              public void onClick(DialogInterface dialog, int which) {
                   dialog.dismiss();
              }
          });
     statDialog.show();
     return true;
```

At ❶, we connect to the app's default shared preferences, and at ❷, we retrieve the number of games won (which we give a default value of `0` in case this is the first run of the program). At ❸, we build an alert dialog to show the user the number of games they've won, and at ❹, we display the number along with an encouraging message.

Save that final change and then build and run your app. Your last challenge may be trying to stop playing! Figure 5-3 shows how the Game Stats screen may look if you have a couple of math whizzes playing your game.

Figure 5-3: The Game Stats screen keeps track of the number of times you (or your friends) have guessed the correct number.

Adding options menus, saving game stats and user preferences, displaying alert dialogs—these are the kinds of finishing touches that can make your game or any other app really stand on its own as a professional-looking, fully functional mobile application. Keep making improvements to your app as you think of new features, and you'll have an app worth sharing with friends—or with the world. Happy coding!

What You Learned

You've built a professional-quality Android mobile game by adding several finishing touches to the Hi-Lo guessing game app, including:

- Adding an options menu to an Android app
- Designing the options menu by editing the menu's XML file
- Displaying the options menu using a `MenuInflater`
- Responding to user selections in a menu
- Using a `switch` statement with multiple `case` statements to replace long `if-else` chains
- Creating customized pop-ups in Android with the `AlertDialog` class
- Storing shared preferences and app statistics using the `SharedPreferences` class
- Retrieving the user's shared preferences when an app starts

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned and to expand your programming skills. Visit the book's website at *https://www.nostarch.com/learnjava/* to download sample solutions.

#1: You Win Some, You Lose Some

Programming Challenge #1 in Chapter 4 (page 106) asked you to give the user seven tries to guess a number between 1 and 100. Now that you've added the ability to change the range, you need to change the number of tries to match.

You learned in Chapter 4 that we could guess a number between 1 and 100 using a binary search strategy (guessing in the middle of the remaining possible values each time) because 2^7, or two to the seventh power, is equal to 128. This means that we should be able to guess a number between 1 and 128 in seven guesses every time using the binary search method. But how many guesses do we need to get a number between 1 and 10, or 1 and 1,000?

To figure out the number of tries needed, we need to know the lowest exponent we can raise 2 to that will give us a number greater than the range. For example, for a number between 1 and 10, $2^4 = 16$, and $16 > 10$, so we need up to four guesses for that range; for a range from 1 to 1,000, $2^{10} = 1{,}024$, so we need

10 guesses.

To find the exponent to which you need to raise a number to be equal to another number, you can use *logarithms*. A logarithm takes a number and a base to find the exponent the base should be raised to in order to result in the given number. Java has a `Math.log()` method that will take a number and find the exponent for base 10. When you divide the logarithm of one number by the logarithm of another number in base 10, the result is the same as taking the logarithm of the first number with the second number as a base. This means that dividing `Math.log(range)` by `Math.log(2)` will tell you what power of 2 will give you `range`. Because exponents can be decimals and you don't want the user to have a noninteger number of guesses, like `7.25`, you'll need to also round up and cast the result into an `int`. To find the exponent for the number of guesses needed for each range, you can use the expression `(int)(Math.log(range)/Math.log(2)+1)`.

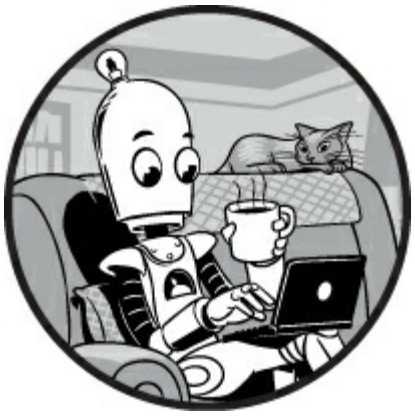
Modify the Hi-Lo guessing game to adapt the maximum number of guesses to suit the range the user selects, both when the game starts and whenever the user selects a new range in the options menu. For example, you could create a variable called `maxTries` to use in place of the hardcoded number `7` when testing whether your user has run out of tries.

#2: Ratio of Wins to Losses

After completing Programming Challenge #1, modify the Hi-Lo guessing game app to store both the number of games won *and* lost. Modify the game stats menu code to retrieve both numbers and show the number of games won, the total number of games played, and the winning percentage (games won divided by total games played, multiplied by 100). Figure 5-4 shows an example.

Figure 5-4: The Game Stats screen showing the percentage of games won

6
DECIPHERING SECRET MESSAGES



You've probably sent secret, encoded notes to your friends to try to hide the messages from your parents or teachers. In this chapter, we'll be doing something similar as we create a Secret Messages app.

Our previous app, the Hi-Lo guessing game, was all about numbers—too high, too low, or just right. By contrast, our Secret Messages app will focus on text. You'll learn to work with text strings and manipulate character values in Java to produce the encoded messages.

The Caesar Cipher

Our app will encode and decode secret messages using the *Caesar cipher*, an algorithm developed more than 2,000 years ago that uses letter substitutions to encode messages.

Figure 6-1: A Caesar cipher disk, showing a key value of 13

The Caesar cipher is named for the Roman emperor Julius Caesar (100–44 BCE). Historians say Caesar liked to encode his private messages, such as notes to his military generals, by "shifting" the letters of the alphabet. For example, the cipher disk shown in Figure 6-1 shifts the alphabet by 13 letters.

In the disk, the outer letters align with the inner letters they should be substituted with, so an *A* becomes an *N*, a *B* becomes an *O*, and so on. This is the cipher that created the following example:

```
Secret messages are so cool!
Frperg zrffntrf ner fb pbby!
```

The first line is the *plaintext*: the original, readable version of the message. The second line is the *ciphertext*, the encoded version of the same message.

Try decoding this message back to its original plaintext by reversing the substitution: find each inner letter and substitute it with the corresponding outer letter.

Not all Caesar ciphers are symmetric like this one. *Symmetric* means that the same process can be used for both encoding and decoding a message. For example, *F* becomes *S* when encoding by adding 13, and *S* becomes *F* when decoding by subtracting 13, and you can use the same disk and *key*—the number used to shift the letters—for both processes. The number to shift the letters by (in this case 13) is called a key because knowing it "unlocks" the cipher. The Secret Messages app we'll build will allow us to use any key value.

Setting Up the Secret Messages App

We'll build the Secret Messages app just like we did the Hi-Lo guessing game: first, we'll create a command line version, then a GUI for the desktop, and, finally, an Android mobile app. The command line version will look fairly simple, as shown in Figure 6-2, but it will allow us to test the secret message algorithm quickly and easily.

Select Command Prompt
C:\Users\Bryson Payne\workspace_ch05\SecretMessages\bin>java SecretMessages
Enter a message to encode or decode:
Secret messages are so cool!
Enter a secret key (-25 to 25):
13
Frperg zrffntrf ner fb pbby!
C:\Users\Bryson Payne\workspace_ch05\SecretMessages\bin>

Figure 6-2: The command line version of the Secret Messages app

As you can see in Figure 6-2, the program asks the user to enter a message to encode or decode, followed by a secret key value. The program then responds with the encoded message. Note that when we start coding the app, the entire message will be encoded, including spaces and punctuation. In the final version, we'll add some logic to encode only letters, as shown in Figure 6-2.

Creating the Secret Messages Project in Eclipse

Let's begin by opening Eclipse. If you still have files open from previous projects, close those files now. To create a new Java Project, go to **File ▸ New ▸ Java Project** and name the project *SecretMessages*. Click **Finish**.

In the Package Explorer pane, expand the *SecretMessages* project folder to see the *src* folder. Right-click the *src* folder and go to **New ▸ Class** to create a new Java source code file, which you'll also name *SecretMessages*. Select the checkbox to create a `main()` method, as shown in Figure 6-3.

Figure 6-3: Create a new Java project, with a new class file called SecretMessages, *and select the checkbox to create a main() method stub.*

Click **Finish**, and you'll see the *SecretMessages.java* file in the main window in Eclipse. Let's get started coding the Secret Messages app!

Beginning to Code SecretMessages.java

At the top of the *SecretMessages.java* file, above the `public class SecretMessages` declaration, add the `import` statement for `java.util.Scanner` so we can ask the user for input:

```
import java.util.Scanner;
public class SecretMessages {
    public static void main(String[] args) {
      ❶ Scanner scan = new Scanner(System.in);
      ❷ System.out.println("Enter a message to encode or decode:");
    }
}
```

Inside the `main()` method, at ❶, we set up a `Scanner` object named `scan`. Then,

at ❷, we prompt the user to enter a message to encode or decode.

After the user prompt, we'll create a `String` variable, called `message`, to accept the line of text the user enters:

```
    System.out.println("Enter a message to encode or decode:");
    String message = scan.nextLine();
```

Then, we get the user's next full line of input, up to an ENTER or RETURN character, using the `nextLine()` method of the `scan` object, and we store it in the string `message`.

So far, the app can ask the user for a message and scan that message into a variable. Now we need to learn how to manipulate the characters inside a string to create an encoded version of the message. Save your file before continuing to the next section.

Messing with Strings

Up to this point, the Secret Messages app looks pretty similar to the Hi-Lo guessing game app. We set up an input scanner, prompt the user for some input, scan the console for the user's response, and then capture the input in a variable. What's going to make the Secret Messages app different is its ability to work with the characters inside the string.

The Caesar cipher command line app will take a lot of steps to complete, so we'll build our app in *iterations*. That means that instead of writing out the whole app all at once and hoping the code works in the end, we'll create our app bit by bit, or one iteration at a time, and test it as we go along so that we'll have a functioning app at each point. Each iteration might not have all the features we need, but we'll eventually get to the complete, full-featured app.

We'll need to use text processing to manipulate the inputted string into an output string. To begin working with strings of text in Java, we'll build a simple message reverser. In other words, we'll take the message the user entered and then give it back to them with the letters in reverse order. So, for example, `"Meet me at the arcade at 5pm"` would become `"mp5 ta edacra eht ta em teeM"`.

First, we create a variable called `output` for the reversed string and set it equal to an empty string:

```
    String message = scan.nextLine();
    String output = "";
```

We'll need a loop to run through the characters in the message. We can choose a `for` loop for convenience, because we know the number of characters in the message. (Later, we'll use the `message.length()` method to tell us how many characters the message contains.) A `for` loop declaration in Java needs to do three things: *initialize* a loop variable, test a *condition* to keep going, and *update* the loop variable before the next iteration. We place semicolons between each of the three parts to separate them. The syntax looks like this:

```
for ( initialization; condition; update ) { body }
```

For example, open JShell and enter this code to create a `for` loop:

```
jshell> for ( int x = 0; x < 10; x++ ) { System.out.println(x); }
```

This loop will print the numbers `0` through `9`. The initialization sets the first looping variable `x` to `0`. The condition tests to make sure `x` is less than `10` before proceeding. Finally, the update adds 1 to `x` after each pass through the loop. `x++` uses a shortcut called the *increment operator* because it increments, or adds 1 to, `x` each time through the loop. `x++` is equivalent to the statement `x = x + 1`. In this example, the `for` loop prints the value of `x` each time through the loop, repeating 10 times and stopping when `x` is no longer less than `10`.

So, to reverse the characters in the message string, we might initialize a variable to the position number, or *index*, of the last character in the string and then loop through the characters from last to first (in reverse order), going backward through the string. In our example, this would look like:

```
    for ( int x = message.length()-1; x >= 0; x-- ) { }
```

The character positions in a string in Java are numbered from `0` (the index of the first character) to the length minus 1. The first character is at index `0`, and the *n*th character in a string is at index (*n* – 1).

In Figure 6-4, the first 10 characters of our sample message are shown. The index of each character in the string is shown below the character. The first letter, `M`, is at index `0`, the first `e` is at index `1`, and so on. Notice that spaces count as characters—there are two spaces shown here, at indexes `4` and `7`. The 10th character in the message is at index `9`, the letter `t`. This continues all the way to the last character in the message, at index `message.length()-1`.

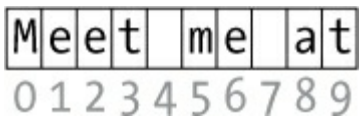


Figure 6-4: The characters in a message labeled with their indexes

We want to start our reversed message at the end of the string, so we initialize `x` to the `message.length()-1`, the index of the last character in the message. The condition is `x >= 0` because we want to keep going all the way down to the first character of the message, at index `0`. Finally, the update is `x--` because we're stepping backward through the string. The opposite of `x++`, `x--` uses the *decrement operator*, which decreases the value of `x` by 1 each time. Come back into Eclipse and begin writing the `for` loop just below the previous line of code:

```
        String output = "";
        for ( int x = message.length()-1; x >= 0; x-- ) {
        }
```

To get the character at a specific position in a string, we use the `charAt()` ("character at") method and give it the index location of the character we want. To append or add a character to a string, we use the `+` operator. Putting those together, we can build the reverse version of the string `message` and store it in the string `output`:

```
      for ( int x = message.length()-1; x >= 0; x-- ) {
          output += message.charAt(x);
      }
```

The body of the `for` loop is a single line. We're getting the character at the `x`th index of `message` and adding it to `output`. Remember to use open and close braces around the body of the `for` loop, because we'll want to add more lines inside the loop as the app grows.

Now we just need to show the output message to the screen, which we can do with `System.out.println(output)`. The full message-reversing iteration of the app is shown in Listing 6-1.

```
import java.util.Scanner;
public class SecretMessages {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a message to encode or decode:");
        String message = scan.nextLine();
        String output = "";
        for ( int x = message.length()-1; x >= 0; x-- ) {
```

```
            output += message.charAt(x);
        }
        System.out.println(output);
    }
}
```

Listing 6-1: The first iteration of the Secret Messages app reverses the characters in the user's message.

You can run the program in Eclipse and test it with your own message, as shown in Figure 6-5. This isn't our finished Secret Messages app, but it does make the message harder to read and easy to decode.

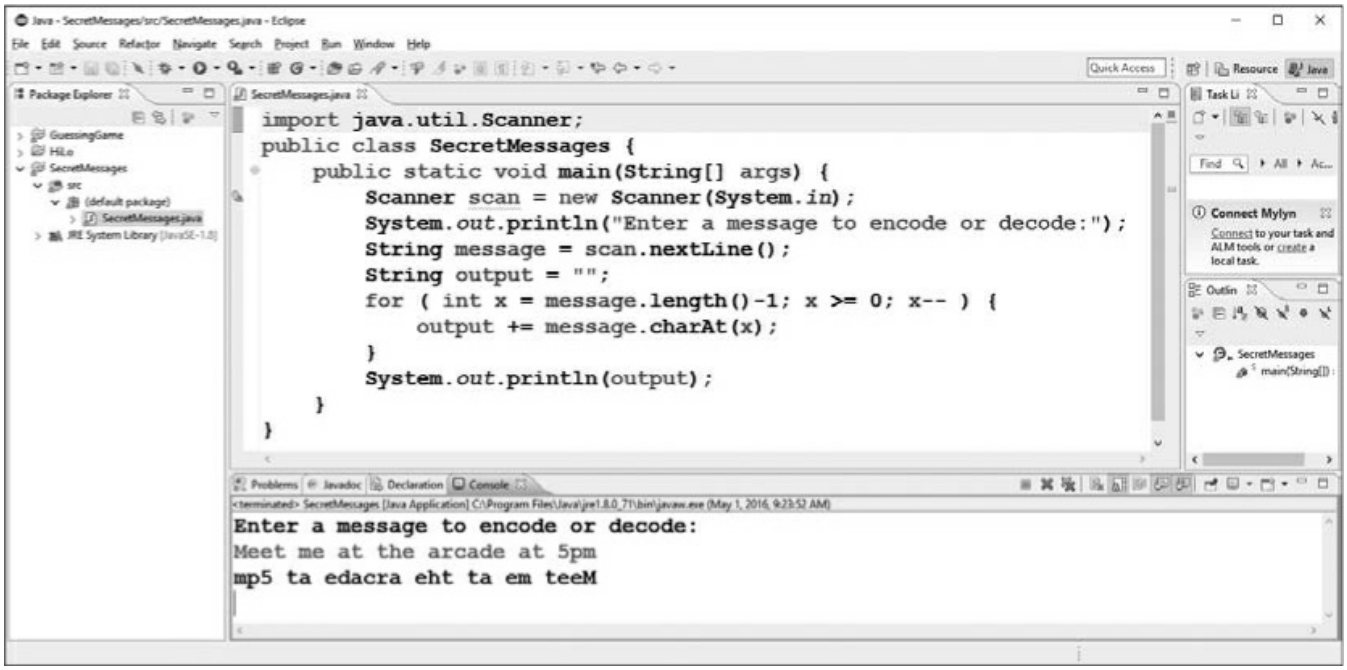

import java.util.Scanner;
public class SecretMessages {
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
System.out.println("Enter a message to encode or decode:");
String message = scan.nextLine();
String output = "";
for (int x = message.length()-1; x >= 0; x--) {
output += message.charAt(x);
}
System.out.println(output);
}
}
Enter a message to encode or decode:
Meet me at the arcade at 5pm
mp5 ta edacra eht ta em teeM

Figure 6-5: Run the program and enter your own message to reverse in the console window at the bottom of the screen.

You can even decode a message by copying the encoded, reversed message and pasting it into the program, as shown in Figure 6-6. So you can type a message, encode it, and paste it into a message to a friend, and then they can decode it by pasting the encoded message into the same program. You've built your first secret message encoder!

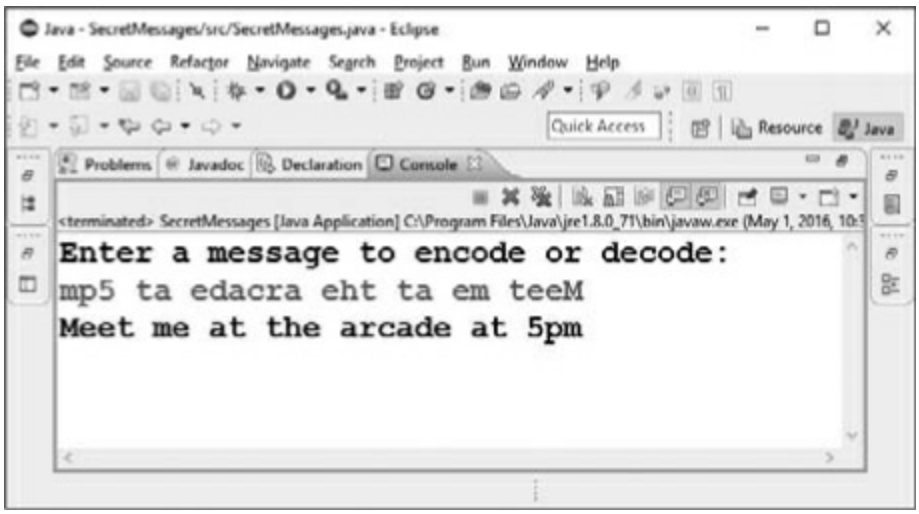

Java - SecretMessages/src/SecretMessages.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
Problems
Javadoc
Declaration
Console
<terminated> SecretMessages [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (May 1, 2016, 10:5
Enter a message to encode or decode:
mp5 ta edacra eht ta em teeM
Meet me at the arcade at 5pm

Figure 6-6: Copy and paste an encoded message into the running program to decode it.

This version of the app is simple enough that you can read the messages without running them through the program, so it's not a very secure way to send messages to your friends. But you did learn how to loop through a string of text using a `for` loop, how to get the string's `length()`, how to access a specific character at a given index or position in the string using `charAt()`, and how to add characters to the end of a string using the `+` operator. In the next section, you'll learn how to change the values of the individual characters in the message to make it harder to read, but still easy for our program to decode.

Characters and Values in Java

Building a better secret message encoder requires the ability to work with character values in strings of text. For the Caesar cipher, we need to be able to shift those values—for example, changing an *A* to an *N*, and an *N* to an *A*. To do this, you need to understand how characters are stored in a computer.

In Java, individual characters, such as `'A'`, can be stored in their own data type: `char`. Notice that we use single quotation marks to contain character values. The `charAt()` method of a string returns a `char` value representing a single character. Java uses 16-bit Unicode characters. *Unicode* is an international character set containing thousands of characters and symbols from all over the world that are represented as numeric values. The `char` type is a way to store Unicode characters, such as `'A'`, `'ñ'`, and `'ç'`, by their numeric values.

We can add to a Unicode `char` value in the same way we can add to an `int` variable. With the Secret Messages app, we want to add the `char` value for `'A'`

(65) to the key (13) to get the new char value (78), which represents the encoded letter ('N').

For our second iteration of the app, let's begin coding the Caesar cipher by setting up a char variable called key and storing the value 13 in it. We'll add this below the String output line:

```
      String output = "";
      char key = 13;
```

We also need to modify the for loop to go from the beginning of the string to the end. We're starting at index 0 this time and moving through the string while x is less than message.length(). Each time we repeat the loop, we're adding 1 to x, the character position. Here is the new for loop:

```
      char key = 13;
      for ( int x = 0; x < message.length(); x++ ) {
```

Finally, instead of adding the original characters of the message string to the output string, we need to add the key value to each character, making sure we get a char value that we can add to the output string:

```
          output += (char)(message.charAt(x) + key);
```

We've added the key value to each character in the message string, and then we've taken that sum and cast it to a char. Putting char in parentheses before an expression forces the value to its right to fit into a char data type, or *casts* the value to that type. We have to cast the output value to (char) here because in Java, the right side of this equation may result in an int value by default. Listing 6-2 shows the program with these changes.

```
import java.util.Scanner;
public class SecretMessages {
    public static void main(String[] args) {
       Scanner scan = new Scanner(System.in);
       System.out.println("Enter a message to encode or decode:");
       String message = scan.nextLine();
       String output = "";
       char key = 13;
       for ( int x = 0; x < message.length(); x++ ) {
           output += (char)(message.charAt(x) + key);
       }
       System.out.println(output);
    }
}
```

Listing 6-2: The Secret Messages app is still short at just 14 lines, but it encodes strings of text now!

If you run the program at this point and enter a message, you should see output like the following:

```
Enter a message to encode or decode:
Secret messages are so cool!
`rp□r?-zr??ntr?-n□r-?|-p||y.
```

The message is encoded, but it's not quite what we were expecting. First of all, we're encoding all characters, including spaces and punctuation marks. Second, we're not wrapping around to the front of the alphabet yet, because we're *always* adding 13 to every character, not taking into account the fact that the letters at the end need to wrap back around to the beginning of the alphabet, resulting in odd symbols and unprintable characters in the output message. We'll cover both of these issues in the next section. Before moving on, save your program.

Encoding Just the Letters

Our second iteration of the Secret Messages app gave us an encoded message, but it lacks some features that we want in the final app. The upgrades needed to create the final app will require some logic in the form of `if` statements and conditions to encode only letters (not spaces or punctuation) and to make the alphabet wrap around during encoding. We'll tackle these improvements in our third iteration.

First, we want to perform some tests on each character in the input message instead of just adding the character directly to the output message. Let's change the body of the `for` loop as follows:

```
    for ( int x = 0; x < message.length(); x++ ) {
        char input = message.charAt(x);
    }
```

The character stored in `input` will be the first character in `message`, then the second, and so on. We need to test each character one by one to check whether it's a character that we want to encode. We want to encode letters by adding the key value to them and wrapping to the front of the alphabet if necessary.

Otherwise, if the character is a space or punctuation, we'll leave the character unchanged and add it to the output message.

Let's add the following `if` statement in the body of our `for` loop:

```
        char input = message.charAt(x);
        if (input >= 'A' && input <= 'Z')
```

In the condition inside the `if` statement, we can use either `65` (the value of `'A'`) or the character literal `'A'` itself. The condition tests whether the character stored in `input` is greater than or equal to `'A'` and less than or equal to `'Z'`—in other words, whether `input` contains an uppercase letter. If this condition is `true`, we want to add the key value to `input` to shift it, producing the Caesar cipher value to substitute for that letter. Notice that this is only for uppercase letters—lowercase letters will require a separate test.

Now, in the body of the `if` statement, we'll add the key value to the `input` character to encode the letter. This is also where we'll handle wrapping around to the front of the alphabet if the key value shifts a letter past `'Z'`. So the code inside the `for` loop will become:

```
    for ( int x = 0; x < message.length(); x++ ) {
        char input = message.charAt(x);
        if (input >= 'A' && input <= 'Z')
        {
         ❶ input += key;
         ❷ if (input > 'Z')
             ❸ input -= 26;
        }
     ❹ output += input;
    }
```

After getting the next character from the message and checking to make sure it's an uppercase letter, we encode the letter by adding the key to it ❶. Then, we check whether adding the key value pushed the letter past `Z` ❷. If so, at ❸ we subtract `26` (the number of letters in the English alphabet) from the encoded `input` value to wrap it back around to the front of the alphabet. Finally, we can add the resulting `input` character to the `output` string ❹.

If you run the program now, you can encode an all-uppercase message as follows:

```
Enter a message to encode or decode:
```

```
SECRET MESSAGES ARE SO COOL!
FRPERG  ZRFFNTRF  NER FB  PBBY!
```

Notice the punctuation and spaces stay the same, and all uppercase letters are shifted by 13 characters, wrapping around so that `S` becomes `F`, and so on.

Encoding the lowercase letters is logically identical to handling the uppercase letters. You can copy and paste the same `if` statement code, but change the `A` and `Z` to lowercase letters. Don't forget to add an `else` before the second `if` statement. Try it yourself; but if you get stuck, the complete code is provided in Listing 6-3 in the next section.

ENCODING MESSAGES IN OTHER LANGUAGES

The version of the program we're writing will only work consistently for basic Latin characters from *A* to *Z* and *a* to *z*. If your preferred language is not English, however, you can adjust the program as long as Unicode has the alphabet or symbols of your language grouped together in a contiguous set, one after the other. You can test for the first and last character of your preferred language just like the English encoder/decoder in this chapter. If your language either isn't provided in a complete set or uses some of the basic Latin alphabet plus certain other characters (like how Spanish uses the ñ and French uses the ç, along with other accented characters), you could add the key to any character as we did in Listing 6-2. Optionally, you could leave spaces in the message by testing with the `Character.isSpace()` method and encode everything except spaces. Play with different methods for the alphabet or language of your choice to find the encoding scheme that works best for you. Change the program and make it do something new—that's the best way to learn!

Closing the Scanner

There's one touch we can add here that you may recall from Chapter 2. The `Scanner` variable `scan` should have a yellow underline, indicating a warning in Eclipse telling you there is a resource leak. When you hover your mouse over

the warning, it pops up with the message `'scan' is never closed`. Remember that we need to close all input/output resources, such as `Scanner` objects, when we're finished using them. To do so, we'll add the command `scan.close()` right after the final `System.out.println` in the `main()` method of the program:

```
    System.out.println(output);
       scan.close();
```

Adding this line removes the resource leak warning. See Listing 6-3 for a fully functional Caesar cipher encoder and decoder with a key value of `13`.

```
import java.util.Scanner;
public class SecretMessages {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a message to encode or decode:");
        String message = scan.nextLine();
        String output = "";
        char key = 13;
        for ( int x = 0; x < message.length(); x++ ) {
            char input = message.charAt(x);
            if (input >= 'A' && input <= 'Z')
            {
                input += key;
                if (input > 'Z')
                    input -= 26;
        }
        else if (input >= 'a' && input <= 'z')
        {
            input += key;
            if (input > 'z')
              input -= 26;
         }
          output += input;
    }
    System.out.println(output);
    scan.close();
   }
}
```

Listing 6-3: This version of the app is a fully functional Caesar cipher encoder and decoder with a key value of 13.

Try it out by first running the program and encoding a message. Then copy the encoded output and run the program again; when prompted, paste the encoded message and press ENTER. The program will respond with the original, decoded message. Here are two sample runs of the program in which I've copied

the encoded output from the first message to use as the input for the second run:

```
Enter a message to encode or decode:
Secret messages are so cool!
Frperg zrffntrf ner fb pbby!
Enter a message to encode or decode:
Frperg zrffntrf ner fb pbby!
Secret messages are so cool!
```

Because the Caesar cipher is symmetric, running the program on an encoded message decodes the message back to its original plaintext. This means you can run this program as it's written in Listing 6-3 to encode a message, send the encoded version to a friend who also has the encoder program, and have them decode your message instantly.

Unfortunately, this means *anyone* running the program (or anyone who figures out the cipher) can decode your messages just as quickly as you can. For our next iteration, let's make the code a little more interesting by allowing the user to set their own key value. Then, you can pick a different key to send messages to different people, or you can use a new key every time.

Adding a Custom Key Value

The secret message encoder works well with a key value of `13`, but what if we want to encode and decode messages using a different key value, such as `3` (the classical Caesar cipher shift) or `5` or `25`?

We'll need to prompt the user for a key value in addition to the message. We can do that by adding a prompt before the line that creates the `char` key variable:

```
        String output = "";
        System.out.println("Enter a secret key (-25 to 25):");
        char key = 13;
```

We're allowing negative key values as a convenience for decoding messages. If you use a key value of `5` to encode your message and send it to a friend, your friend can use a key value of `-5` to decode the message.

When the user responds with their desired secret key value, we can scan the line of input they provide, parse the integer value out of the line of input, and store the value in an `int` variable:

```
        String output = "";
```

```
System.out.println("Enter a secret key (-25 to 25):");
int keyVal = Integer.parseInt(scan.nextLine());
```

Finally, instead of using `13` as the `key` variable, we'll change the variable key to the following:

```
int keyVal = Integer.parseInt(scan.nextLine());
char key = (char) keyVal;
for ( int x = 0; x < message.length(); x++ ) {
```

Here, we're casting the `keyVal` entered to a `char` and storing that value in the variable `key`, since we can't store an integer directly into a `char`.

To handle negative key values for the decoding process, we'll need to make a change to the logic inside the `if` statements. We have to check whether *subtracting* a value (or adding a negative key) would shift the letter past the beginning of the alphabet (or less than `'A'`). If that happens, we can shift the letter back into the range from `A` to `Z` by *adding* `26`. Add the following code to the `if` statement for uppercase letters:

```
if (input >= 'A' && input <= 'Z')
{
   input += key;
   if (input > 'Z')
       input -= 26;
   if (input < 'A')
       input += 26;
}
```

If we input a negative `key`, we check whether we shifted before `A` and add `26` to wrap back around to the end of the alphabet.

Remember to do the same for the lowercase logic inside the `else-if` statement that follows:

```
else if (input >= 'a' && input <= 'z')
{
    input += key;
     if (input > 'z')
         input -= 26;
     if (input < 'a')
         input += 26;
}
```

After these changes, you'll be able to run the program to encode and decode messages with your own, custom key. Your program should work like the

following example:

```
Enter a message to encode or decode:
You've written a really cool app in Java!
Enter a secret key (-25 to 25):
7
Fvb'cl dypaalu h ylhssf jvvs hww pu Qhch!
Enter a message to encode or decode:
Fvb'cl dypaalu h ylhssf jvvs hww pu Qhch!
Enter a secret key (-25 to 25):
-7
You've written a really cool app in Java!
```

The first time we ran the program, we used a key of `7`. So, to decode the secret message, we ran the program again with the key value `-7`, revealing the original plaintext message.

Go ahead and try it out!

Encoding Digits

Now we have encoded letters, but if we encode a message like the following, any digits remain unencoded, in their original plaintext:

```
Enter a message to encode or decode:
Meet me at the arcade at 5pm.
Enter a secret key (-25 to 25):
8
Ummb um ib bpm izkilm ib 5xu.
```

Notice the `5` in `5pm` remains a `5` in the output message. If we want to encode numbers as well as letters, we need to add one more section of logic to the `for` loop. We'll need to check whether a character falls between `0` and `9` and then encode that character as a different digit. We'll also have to remember to wrap back around to the front or back of the set of 10 digits. We'll have to handle this differently from wrapping around in the set of 26 letters since we're dealing with numbers now.

First, let's add another `else-if` statement right after the `else-if` that handles lowercase letters:

```
        else if (input >= 'a' && input <= 'z')
        {
            input += key;
            if (input > 'z')
```

```
            input -= 26;
        if (input < 'a')
            input += 26;
    }
    else if (input >= '0' && input <= '9')
    output += input;
```

This part looks similar to the previous two `if` conditions, except that we're using the digits `'0'` through `'9'`, instead of the letters `'A'` to `'Z'`, as the range.

The next line, inside the brace for the `if` statement, looks a bit different:

```
    else if (input >= '0' && input <= '9')
    {
        input += (keyVal % 10);
```

First, we're using the *integer* version of the secret key, `keyVal`, which the user entered earlier in the program. Second, we've used the modulo operator (`%`) to keep the shift value for digits between `-10` and `+10`.

We also need to test whether adding the `keyVal` pushed a digit past `9` or before `0`, just as we checked whether encoding a letter pushed it past `Z` or before `A`. But, instead of subtracting `26` to wrap back around to the beginning of the alphabet, we need to subtract `10` to keep a digit between `0` and `9`:

```
else if (input >= '0' && input <= '9')
        {
            input += (keyVal % 10);
            if (input > '9')
                input -= 10;
            if (input < '0')
                input += 10;
        }
        output += input;
```

If encoding a digit pushes it past `9`, we subtract `10`. And, if decoding moves a digit below `0`, we add `10`.

Now we can encode both numbers and letters in the messages we send:

```
Enter a message to encode or decode:
Meet me at the arcade at 5pm and bring $2 to play Pac-Man :)
Enter a secret key (-25 to 25):
7
Tlla tl ha aol hyjhkl ha 2wt huk iypun $9 av wshf Whj-Thu :)
```

The `5` in `5pm` is correctly shifted by seven and wraps back around to `2wt` in the encoded message. The `2` in `$2` is shifted by seven to `$9`. You can test the

preceding secret message by decoding it with a key of -7, and you should get the original plaintext message back, including the numbers.

The full, final text-based version of the Secret Messages app is given in Listing 6-4.

```
import java.util.Scanner;
public class SecretMessages {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a message to encode or decode:");
        String message = scan.nextLine();
        String output = "";
        System.out.println("Enter a secret key (-25 to 25):");
        int keyVal = Integer.parseInt(scan.nextLine());
        char key = (char) keyVal;
        for ( int x = 0; x < message.length(); x++ ) {
            char input = message.charAt(x);
            if (input >= 'A' && input <= 'Z')
            {
                input += key;
                if (input > 'Z')
                    input -= 26;
                if (input < 'A')
                    input += 26;
            }
            else if (input >= 'a' && input <= 'z')
            {
                input += key;
                if (input > 'z')
                    input -= 26;
                if (input < 'a')
                    input += 26;
            }
            else if (input >= '0' && input <= '9')
            {
                input += (keyVal % 10);
                if (input > '9')
                    input -= 10;
                if (input < '0')
                    input += 10;
            }
            output += input;
      }
        System.out.println(output);
        scan.close();
    }
}
```

Listing 6-4: The finished command line version of the Secret Messages app

This Secret Messages app can be a fun way to share scrambled messages back and forth with friends. But some of your friends may not have Eclipse or the Java JDK installed on their computer. Wouldn't it be nice to be able to share the Secret Messages app with them too? In the next section, we'll see how to run a command line Java app without opening Eclipse.

Running Command Line Apps Without Eclipse

We've built two command line apps so far, but we've always run them from inside Eclipse. Eclipse provides a convenient console emulator to let us see how a command line app would look and feel, but what if we wanted to run the app from an actual command line, such as the Command Prompt in Windows or the Terminal on macOS? Or what if we wanted to send one of our command line apps to a friend who doesn't have Eclipse installed on their computer?

Fortunately, most people have at least the JRE, or Java Runtime Environment, installed on their computers.

Finding Your Workspace Folders

To run the app you've written and compiled in Eclipse, first start by finding your Eclipse *workspace* folder. Open it in File Explorer (or Finder), as shown in Figure 6-7.

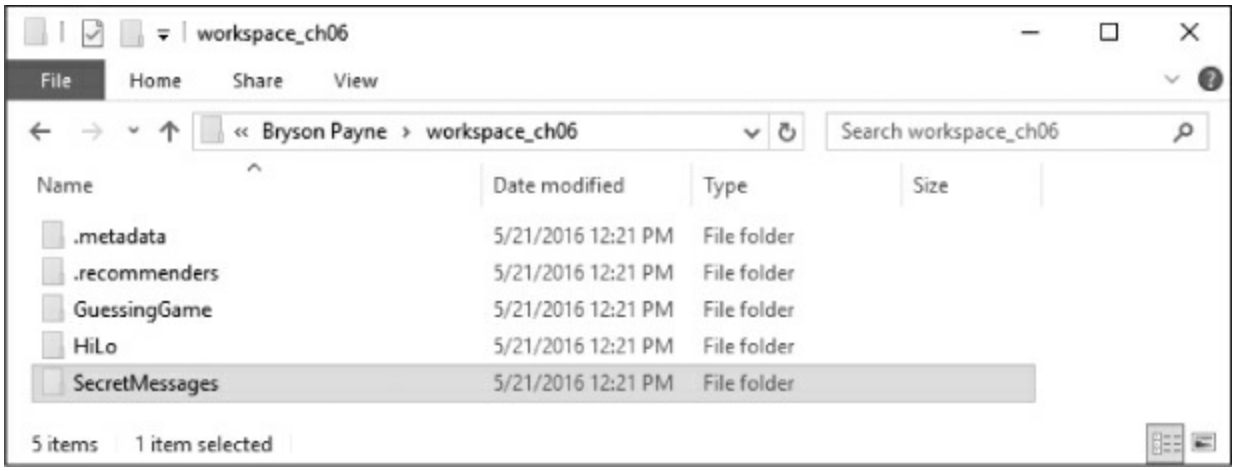


Figure 6-7: My workspace *folder has folders inside it for each of the projects we've built so far, plus some Eclipse default folders.*

Open the *SecretMessages* project folder inside your workspace, and you'll see a

few files and folders inside it, as shown in Figure 6-8. The *src* folder has your source code files (ending in *.java*), and the *bin* folder has the compiled version of your app (ending in *.class*).

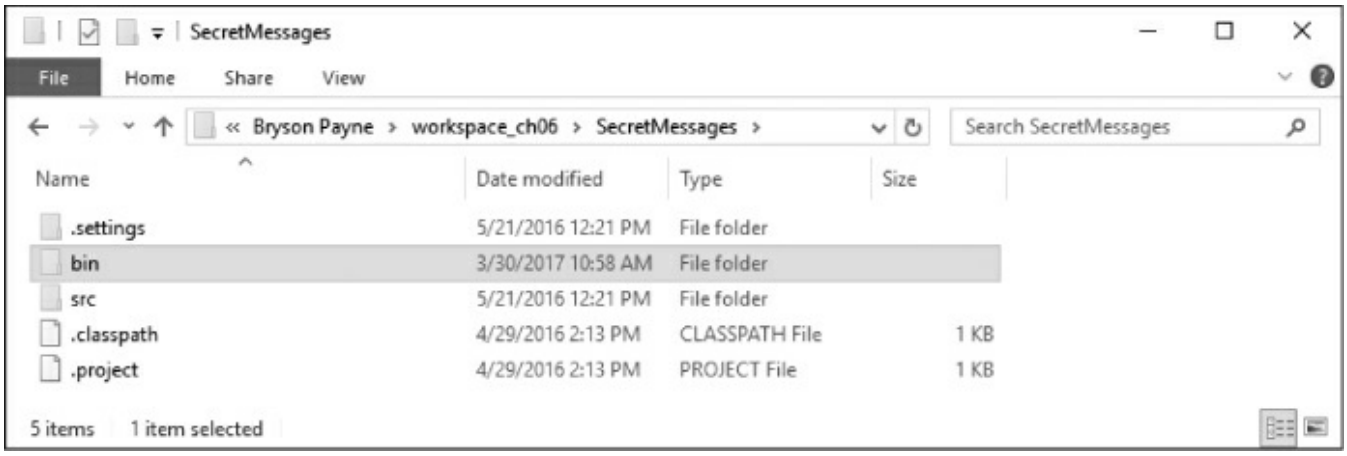


Figure 6-8: Find the bin *folder inside the* SecretMessages *project folder.*

Opening a Command Line Window

Next, open a command line window. On Windows, you can do this by pressing the **Start** button, going to the search bar, and typing **`cmd`** (short for *command*) and pressing ENTER. On macOS, use the Spotlight search bar to find Terminal, or from a new Finder window, go to **Applications ▸ Utilities ▸ Terminal**. On Linux, open the Terminal application or use Search to find Terminal.

We need to browse into the *bin* folder described earlier from the command line or Terminal prompt. Type **`cd`** ("change directory") followed by a space in your terminal window. Then, to change directories into the *bin* folder for your app, drag the *bin* folder from your File Explorer or Finder window and drop it into the Command Prompt or Terminal window, as shown in Figure 6-9.

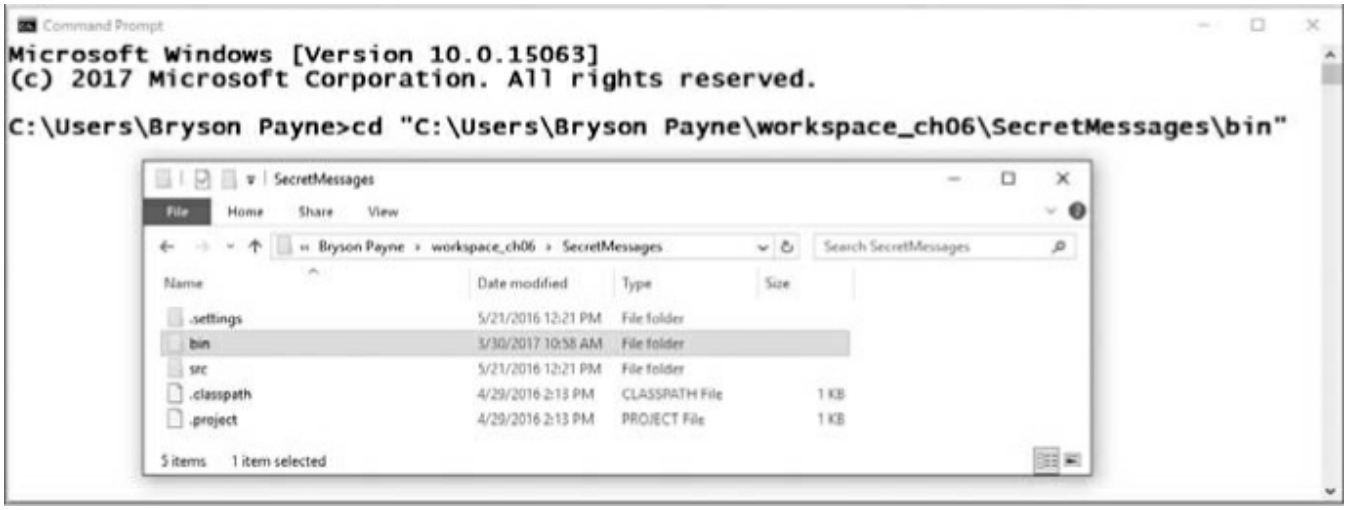


Figure 6-9: You can drag and drop the bin *folder from File Explorer or Finder into the command line window to easily copy the directory path.*

Notice that the full path appears after the `cd` command and looks something like the following (your *workspace/SecretMessages/bin* folder will be in a slightly different location from mine, of course):

```
cd "C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin"
```

On macOS, the command will be:

```
cd /Users/BrysonPayne/Desktop/workspace_ch06/SecretMessages/bin
```

Press ENTER after this command, and the terminal will change the prompt to show you that you're now inside the *bin* folder. On Windows, this looks like:

```
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>_
```

Now we're inside the *bin* folder, where the file *SecretMessages.class* has been compiled. To run the program contained in the *SecretMessages.class* bytecode file, enter the following command:

```
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>java SecretMessages
```

Spelling, capitalization, and spacing all matter in the command, so you'll need to match the name you gave your Java class precisely. If you typed the command correctly, the app should run, and you can test it with a message and secret key like so:

```
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>java SecretMessages
Enter a message to encode or decode:
```

```
I'm running my app directly from the command line terminal!
Enter a secret key (-25 to 25):
12
U'y dgzzuzs yk mbb pudqofxk rday ftq oayymzp xuzq fqdyuzmx!
```

You can run the app again by retyping the line **`java SecretMessages`** or by pressing the up arrow key on your keyboard and pressing ENTER. Figure 6-10 shows a couple of runs encoding and decoding a message from a Windows command prompt.

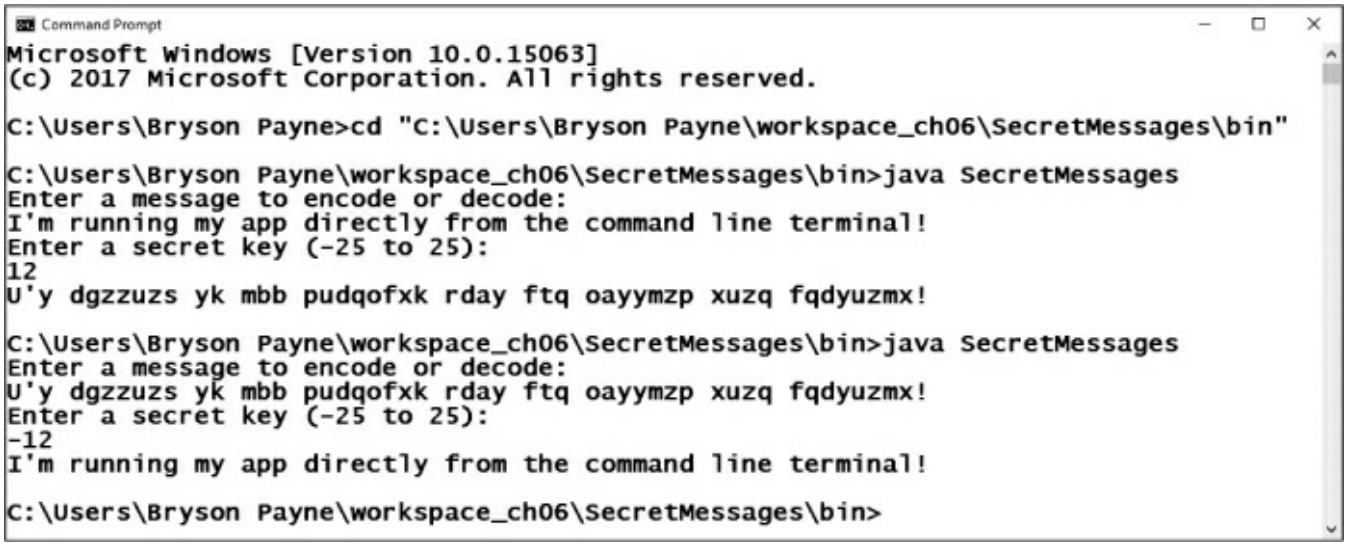

Command Prompt
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.
C:\Users\Bryson Payne>cd "C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin"
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>java SecretMessages
Enter a message to encode or decode:
I'm running my app directly from the command line terminal!
Enter a secret key (-25 to 25):
12
U'y dgzzuzs yk mbb pudqofxk rday ftq oayymzp xuzq fqdyuzmx!
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>java SecretMessages
Enter a message to encode or decode:
U'y dgzzuzs yk mbb pudqofxk rday ftq oayymzp xuzq fqdyuzmx!
Enter a secret key (-25 to 25):
-12
I'm running my app directly from the command line terminal!
C:\Users\Bryson Payne\workspace_ch06\SecretMessages\bin>

Figure 6-10: Now that you can run your Java app from the command line, you can encode and decode messages on any computer with Java installed, with or without Eclipse.

Now that you know how to run the app on your computer, you can share the *SecretMessages.class* file with your friends to exchange encoded messages. If their computer has Java (JDK or JRE) installed, all they'll have to do is open their command line program, change directories into the folder containing the *SecretMessages.class* file, and run the same **`java SecretMessages`** command. To exchange messages, just agree on a key value to use. You can choose to use the same key every time or use different key values with different friends! Remember, though, this message app is just for fun—anyone with this program, or with a little time on their hands, can break a simple Caesar cipher. We'll see how to break Caesar ciphers easily in Chapter 7.

What You Learned

The Secret Messages app provided a fun way to delve into the world of

manipulating characters and strings of text in Java. Among the new things you learned in this chapter were:

- Using the Caesar cipher to encode and decode simple messages
- Understanding the `char` data type for storing single Unicode characters
- Getting a specific character in a string with the `charAt()` method
- Accessing a specific location in a string using the index, or position number, of a character in the string
- Adding strings and characters together using the `+` operator
- Looping through a string with a `for` loop, using its `length()` function to get the number of characters
- Understanding how the computer stores characters and other data represented by number values
- Running command line apps directly from the command line, without Eclipse

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned, as well as to expand your programming skills by trying something new. Visit the book's website at *https://www.nostarch.com/learnjava/* to download sample solutions.

#1: Looping the Loop

We've built a fun message encoder/decoder app that will let us send and receive secret texts, emails, tweets, and so on. For your first programming challenge, add a loop to the Secret Messages app that will allow you to keep encoding and decoding messages as many times as you like.

You can choose to use the same key every time, or you can ask the user for a new key each time. If you ask the user to enter a secret key each time through the loop, you can enable them to encode and decode messages, one after the other (entering `8`, for example, to encode a message to a friend, and `-8` to decode a message from that same friend).

One way to do this is to prompt the user to enter a new message to encode or

decode or tell them to press ENTER to quit. Then, use an `if` statement to check whether the user entered a message or just pressed ENTER.

Give it a shot!

HINT

message.length() will be greater than zero if the user typed something.

#2: Reversing and Encoding

Let's make our messages even harder to decipher by reversing the message before we encode it with the Caesar cipher. This double-encoding version of the app will combine the message-reversing approach of the first version in Listing 6-1 on page 130 with the Caesar cipher features of the final version from Listing 6-4 on page 140.

Reversing the order of the message is another example of symmetric encryption—doing it once reverses the message, and doing it again restores the message. Combining the string reversal with the Caesar cipher shift doesn't make the program much more difficult, but it can make your messages even more confusing to an eavesdropper.

Make sure your app correctly encodes and decodes messages, and remember that your friends will need the new version of the program to be able to decipher your new, doubly scrambled messages! Enjoy!

#3: Safely Handling Keys with try and catch

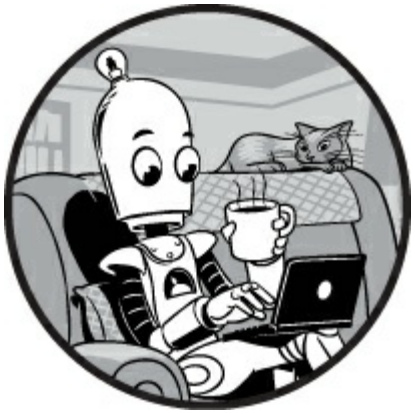
A final improvement you could make to the Secret Messages app would be to safely handle bad numeric input when the user enters a key value. Remember the `try-catch` statement?

Adding the `try-catch` blocks to this app can be a bit tricky. In addition to adding the exception-handling code to prevent the user from crashing the program by giving a bad number (for example, by entering text instead of a number when prompted for the key value), you'll need to think about what the app should do when bad input is provided. Should it tell the user their key was invalid and use a predefined key instead (say, `13`)? Or should it create a loop that will keep asking the user for a valid key?

Try coding it both ways and pick the one you like best!

7

CREATING ADVANCED GUIs AND SHARING YOUR APP



For the GUI desktop version of the Secret Messages app, we're going to provide two large text areas that allow the user to copy and paste long messages in the GUI interface. By the end of the chapter, we'll add a slider for key selection like the one shown in Figure 7-1, which will make cracking Caesar ciphers very easy, even when the key is unknown!

Dr. Payne's Secret Message App
Qxf lxdum cqn bnlanc vnbbjpnb jyy pnc nenw lxxuna?
Jmm j wrln PDR frcq fxam fajy, jwm j burmna cqjc
uncb hxd nwlxmn, mnlxmn jwm nenw lajlt vnbbjpnb!
Cqjc fxdum kn jfnbxvn - jwm rc'b rw Lqjycna 6!
Key:
-9
Encode/Decode
-26
-13
0
13
26
How could the secret messages app get even cooler?
Add a nice GUI with word wrap, and a slider that
lets you encode, decode and even crack messages!
That would be awesome - and it's in Chapter 7!

Figure 7-1: The Secret Messages encoder/decoder app we'll build in this chapter

This version of the app is more user-friendly than the console-based app, and we'll be able to reuse some code from Chapter 6 because the algorithm for encoding and decoding messages is the same.

Setting Up the Secret Messages GUI App Project

Open Eclipse and create a new Java project by going to **File ▸ New ▸ Java Project**. Name the project **`SecretMessagesGUI`** and then click **Finish**. Close any open files and expand the *SecretMessagesGUI* project folder in the Package Explorer pane to find the *src* folder. Right-click the *src* folder and select **New ▸ Class** to create a new Java source code file. Name this **`SecretMessagesGUI`** as well.

We're going to use the Swing toolkit again, so in the New Java Class dialog, change the superclass to `javax.swing.JFrame` and select the checkbox to create a `main()` method.

Click **Finish**, and you'll see some familiar skeleton code for the *SecretMessagesGUI.java* file. Right-click the *SecretMessagesGUI.java* file in the Package Explorer and select **Open With ▸ WindowBuilder Editor** to begin creating the GUI interface for the Secret Messages app.

Designing the GUI and Naming Components

Open the WindowBuilder Editor's Design tab. Expand the `javax.swing.JFrame` component in the Components pane and select **getContentPane()**. Now go to the Properties pane and change the `Layout` property to **Absolute layout**, as shown in Figure 7-2. This will allow us to position components with pixel-perfect accuracy.

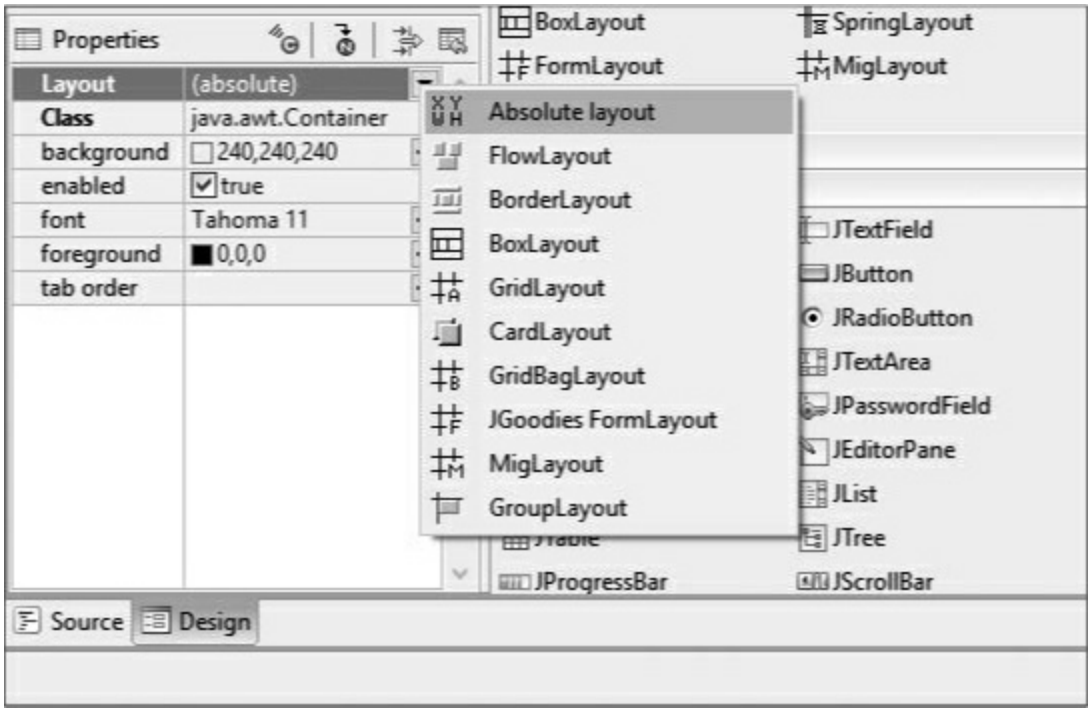


*Figure 7-2: Change the Layout property to **Absolute layout** before placing any GUI components.*

Next, in the Components pane, click **javax.swing.JFrame**. Then, in the Properties pane, set the `defaultCloseOperation` to **EXIT_ON_CLOSE**. Also, change the `title` property to ***Your Name*'s Secret Message App**, as shown in Figure 7-3.

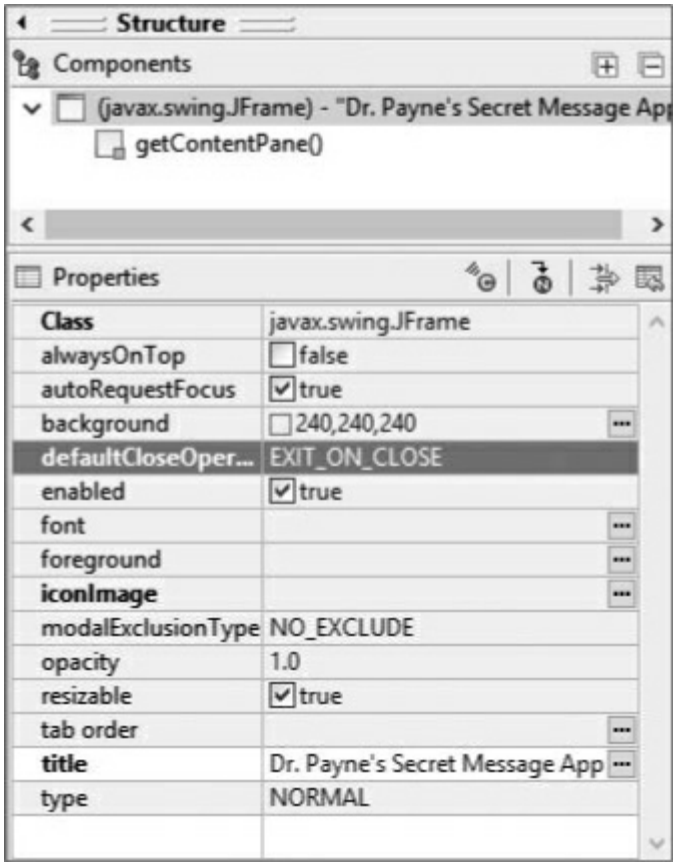


Figure 7-3: Change the defaultCloseOperation and title properties to customize the app.

Next, let's give ourselves some room to build a slightly larger GUI interface so we can encode and decode longer messages. With `javax.swing.JFrame` still selected in the Components pane, click the outside border of the window in the design preview. Click the small black resize box at the lower-right corner of the window, and drag down and to the right to resize the frame to 600 by 400 pixels, as shown in Figure 7-4.

Figure 7-4: Resize the JFrame to hold bigger messages.

Now we're ready to begin placing the GUI interface components. For the GUI version of the app, we want to be able to encode and decode long messages, so the first component we'll place is a `JTextArea`. Like a `JTextField`, text areas allow you to enter text, but a `JTextArea` can hold, display, and even wrap several lines of text.

To insert the first `JTextArea` for the user to input a message, under Components in the Palette, click **JTextArea**. Then place it near the top of the `JFrame` in the design preview. Click and drag to draw a `JTextArea` approximately one-third of the height of the content pane, with a small margin between the `JTextArea` and the edge of the frame. The `JTextArea` should be approximately 564 pixels wide by 140 pixels tall. Then, in the Properties pane, change the `Variable` property to **`txtIn`**.

To create the second `JTextArea` for the outputted message, right-click **txtIn** in the Preview pane, select **Copy**, right-click the content pane, and select **Paste**. Place the new `JTextArea` near the bottom edge of the frame. To name the new `JTextArea`, change the `Variable` property to **`txtOut`**. You should end up with two

`JTextArea` components, as shown in Figure 7-5.

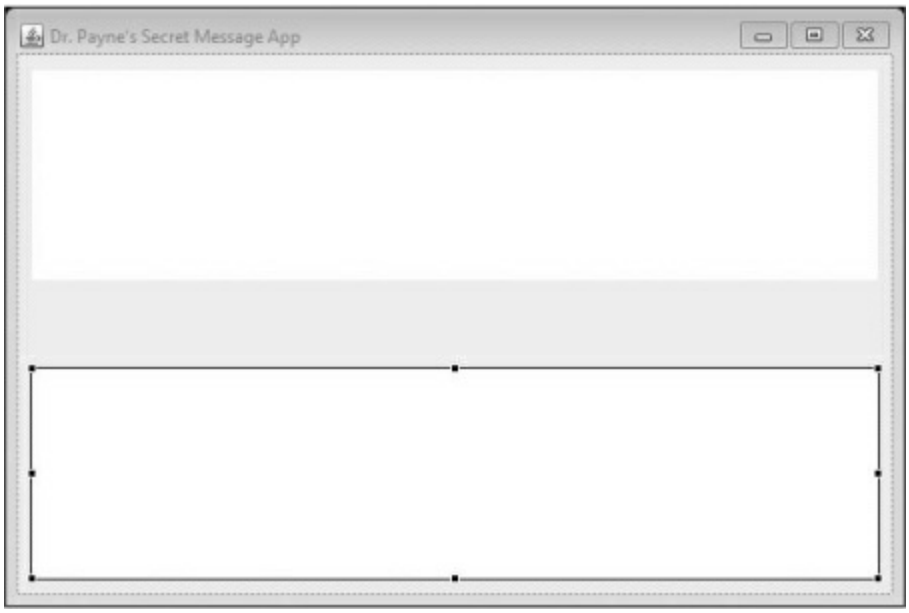


Figure 7-5: Copy and paste txtIn to create a second JTextArea and name it txtOut.

Next, let's add a text field for the encryption key value. Click **JTextField** in the Palette pane under Components and then click near the center of the content pane to place the `JTextField`. Change the `JTextField`'s `Variable` property to **`txtKey`** and resize it to about half its original size.

Add a `JLabel` to the left of `txtKey`, entering **`Key:`** as the label's `text` property. Set the `horizontalAlignment` to **RIGHT**. Finally, add a `JButton` to the right of `txtKey`, entering the words **`Encode/Decode`** as the button's `text` property. Expand the width of the button until it shows all the text. The interface should look like Figure 7-6. To preview the interface without running the app, press the small Test GUI button above the Palette, circled in Figure 7-6.

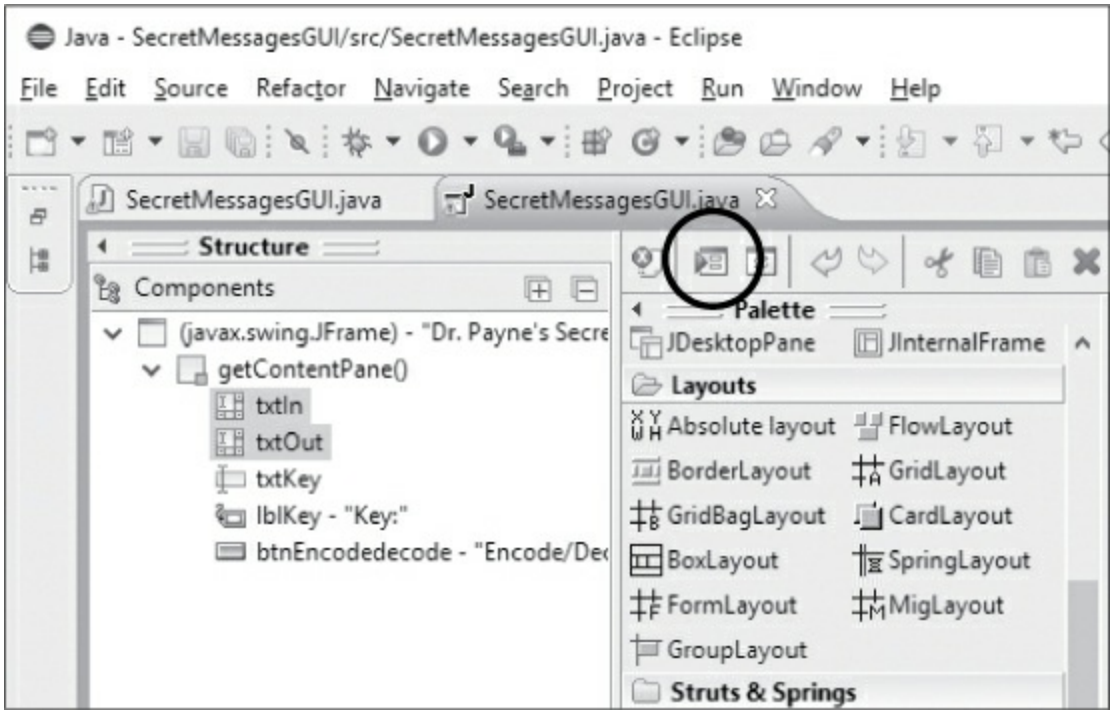


Figure 7-6: Test the GUI interface by pressing the Test GUI button above the Palette.

Now that you have the user interface set up, it's time to code the program.

Coding the Secret Messages GUI App

Switch back to the source view by clicking the Source tab on the bottom left of the design view window. You'll see that Eclipse has added all the GUI code in the *SecretMessagesGUI.java* file. To connect the GUI to our app, we first need to add the declarations for the two `JTextArea` variables at the top of the `SecretMessagesGUI` class. Eclipse knows that `JTextField` components usually need event handlers for user input, so it places `txtKey` at the top of the class, but we need to do the same with the two `JTextArea` components, `txtIn` and `txtOut`. Add the two bottom lines shown here:

```
public class SecretMessagesGUI extends JFrame {
    private JTextField txtKey;
    private JTextArea txtIn;
    private JTextArea txtOut;
```

After declaring the `JTextArea` variables at the top of the file, we need to modify the code inside the `SecretMessagesGUI()` constructor. Delete the variable type `JTextArea` at the beginning of each of these two lines:

```
public SecretMessagesGUI() {
    setTitle("Dr. Payne's Secret Message App");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().setLayout(null);
    txtIn = new JTextArea(); // Delete "JTextArea" from start of line
    txtIn.setBounds(10, 11, 564, 140);
    getContentPane().add(txtIn);
    txtOut = new JTextArea(); // Delete "JTextArea" from start of line
    txtOut.setBounds(10, 210, 564, 140);
    getContentPane().add(txtOut);
```

With these changes, we're ready to begin writing the method to encode messages whenever the user clicks the Encode/Decode button.

Creating the encode() Method

The `encode()` method we'll write is similar to `checkGuess()` from the Hi-Lo guessing game app in Chapter 3. The `checkGuess()` method took a string that was the user's guess, but the method didn't need to return a value, so it returned `void`. Unlike `checkGuess()`, `encode()` will take a message and an encryption key as values, and instead of returning `void`, it will return an encoded message.

Methods return values so that the program can use them. We want `encode()` to take a message and key from the user and then run some code on those values to produce the encoded message, which our program will use as the output for the user. When declaring a method that returns information, we state the return value's data type before the name of the method. The type of information we want returned from `encode()` is a `String` variable. So, `encode()` will be declared as `public String encode()`.

We need to tell Java what kind of information we want to pass to `encode()`; this information is known as a method's *parameters*. To declare parameters for a method, we place them inside the parentheses following the method name, with the data type first and the name of the parameter second. Multiple parameters are separated by commas. The declaration for the `encode()` method looks like this:

```
public String encode( String message, int keyVal )
```

Let's add braces for the body and place the `encode()` method right below the declarations for the two `JTextArea` variables, above the `SecretMessagesGUI()` constructor method:

```
public class SecretMessagesGUI extends JFrame {
    private JTextField txtKey;
    private JTextArea txtIn;
    private JTextArea txtOut;
    public String encode( String message, int keyVal ) {
    }
    public SecretMessagesGUI() {
```

Eclipse will underline the `encode()` method in red to let you know you haven't returned a value, but that's because we haven't written the code inside the method body yet. Here we'll be reusing some of our code from Chapter 6.

Declare a `String` called `output`, set it equal to an empty string for now, and add a line to return `output` as the result of the `encode()` method using a `return` statement:

```
  public String encode( String message, int keyVal ) {
      String output = "";
      return output;
  }
```

Since `encode()` was declared to return a `String` value and we're returning `output`, which *is* a `String`, Eclipse will remove the red underline to let us know we've fixed the problem of the missing return value.

Next, we'll reuse a selection of code from the text-based version of the app for encoding messages. Open the *SecretMessages* project from Chapter 6 in the Project Explorer pane.

We can copy the code starting at `char key` and ending at the closing brace of the `for` loop, right before `System.out.println(output);` near the end of the program. This is the part of the app that encodes a message using the Caesar cipher algorithm.

Paste the copied code from *SecretMessages.java* into *SecretMessagesGUI.java*, inside `encode()`, between the two lines we just added. The finished `encode()` method looks like this:

```
public String encode( String message, int keyVal ) {
    String output = "";
    char key = (char) keyVal;
    for ( int x = 0; x < message.length(); x++ ) {
        char input = message.charAt(x);
        if (input >= 'A' && input <= 'Z')
        {
            input += key;
```

```
            if (input > 'Z')
                input -= 26;
            if (input < 'A')
                input += 26;
        }
        else if (input >= 'a' && input <= 'z')
        {
            input += key;
            if (input > 'z')
                input -= 26;
            if (input < 'a')
                input += 26;
        }
        else if (input >= '0' && input <= '9')
        {
            input += (keyVal % 10);
            if (input > '9')
                input -= 10;
            if (input < '0')
                input += 10;
        }
        output += input;
    }
    return output;
}
```

Because we named our variables consistently, we can reuse the Caesar cipher encryption code. Consistent naming is a good habit, and well-written code in Java can be reused across multiple platforms.

Writing the Event Handler for the Encode/Decode Button

We want the user to provide both the input message and the key value that we'll pass along to `encode()` in the GUI interface, and we want the output message to be returned when the user presses the Encode/Decode button, so let's write the code to handle the button click event.

Switch to the design view in *SecretMessagesGUI.java* and double-click the **Encode/Decode** button. Eclipse will switch back to the source view and insert the code for the `actionPerformed()` event handler for the Encode/Decode button, like the code snippet shown here:

```
    JButton btnEncodedecode = new JButton("Encode/Decode");
    btnEncodedecode.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
        }
    });
```

This is an anonymous inner class, like the one we first encountered in Chapter 3. All we have to add is the code inside `actionPerformed()` to tell Java how the button should behave when it's clicked.

We want the Encode/Decode button to perform the following steps:

1. Get the input message from `txtIn`.
2. Get the key from `txtKey`.
3. Encode the message using the key.
4. Show the output message in `txtOut`.

Think about how we might accomplish each of those steps. Try it yourself before reading further and then read below to see how my solution compares.

For the first step, we can set up a `String` called `message` that gets the text from the `txtIn` field. Just like a `JTextField`, a `JTextArea` has a `getText()` method:

```
        public void actionPerformed(ActionEvent arg0) {
            String message = txtIn.getText();
```

For the second step, we can get the user's key from `txtKey` using the `Integer.parseInt()` method, which pulls an integer value out of a text string. Then we'll store the integer we get in a variable we'll call `key`:

```
            String message = txtIn.getText();
            int key = Integer.parseInt( txtKey.getText() );
```

To encode the message in the third step, we just need to call the `encode()` method and pass it two arguments. *Arguments* are the values we give a method to serve as its parameters. Remember we defined `encode()` to require two parameters, so we'll need to pass `encode()` two arguments, which will be the variables `message` and `key`.

We can call `encode()` with the statement `encode(message, key)`. When it has finished executing, it will return the encoded message, which is `output`. Note that a method returns the value itself, not the variable the value is stored in. That means that the encoded message is stored in `output`, but when `encode()` returns `output`, we're actually getting only the encoded string. In addition to that, all the code and variables created inside `encode()` aren't carried over to the rest of the program, so the `output` variable doesn't exist anymore once `encode()` returns. We'll need to store the value returned by the method in a new variable if we

want to save it. In order to keep our naming consistent, let's call the new variable `output` too:

```
        String message = txtIn.getText();
        int key = Integer.parseInt( txtKey.getText() );
        String output = encode( message, key );
```

Finally, we'll use the `setText()` method to show the `output` message in `txtOut`:

```
        String message = txtIn.getText();
        int key = Integer.parseInt( txtKey.getText() );
        String output = encode( message, key );
        txtOut.setText( output );
```

The `actionPerformed()` method is almost complete, except for some error handling. We've used the `Integer.parseInt()` method, which can throw exceptions if the user enters bad input, so we'll need to add `try-catch` statements.

Handling Bad Input and User Errors

The code to process button clicks includes a call to `Integer.parseInt()`, and we learned back in Chapter 3 that this method can throw exceptions when the user enters erroneous input. In particular, if the user were to leave the `txtKey` field blank or type anything other than an integer into that `JTextField`, the `Integer.parseInt()` method would fail.

We need to protect the program from bad input by using `try-catch` blocks to handle exceptions. Since we want to encode or decode a message only when the user has correctly entered an encryption key, we can wrap all four lines of `actionPerformed()` inside a `try` block by adding the keyword `try` and an opening brace before the first line, as well as a closing brace after the fourth line, as shown here:

```
        public void actionPerformed(ActionEvent arg0) {
            try {
                String message = txtIn.getText();
                int key = Integer.parseInt( txtKey.getText() );
                String output = encode( message, key );
                txtOut.setText( output );
            } catch (Exception ex) {
            }
        }
```

Eclipse will highlight the closing brace in red, so let's go ahead and add the

`catch` block as shown. We'll need to tell Java what to do when bad input is received, and we'll add those steps inside the braces for the `catch` statement a little later. But first, let's finish the `main()` method so that we can test the app.

Building the main() Method and Running the App

Just like when we created our game app in Chapter 3, we need to add some setup code to the `main()` method to create an instance of the Secret Messages app, correctly size the GUI window, and make the GUI visible to the user. Near the bottom of the *SecretMessagesGUI.java* file, find the `main()` method stub Eclipse provided and add the following three lines.

```
public static void main(String[] args) {
  ❶ SecretMessagesGUI theApp = new SecretMessagesGUI();
  ❷ theApp.setSize(new java.awt.Dimension(600,400));
  ❸ theApp.setVisible(true);
}
```

The line at ❶ creates a `SecretMessagesGUI` object called `theApp`. The `new` keyword followed by the class name runs the constructor method `SecretMessagesGUI()`, which sets up all the components of the GUI interface.

Next, we set the size of the `JFrame` to match the width and height that we used in designing the layout, 600 by 400 pixels ❷. Finally, we set the `JFrame`'s `visible` property to `true` ❸, allowing the user to see the GUI. Save these changes and then run the app to test it, as shown in Figure 7-7.

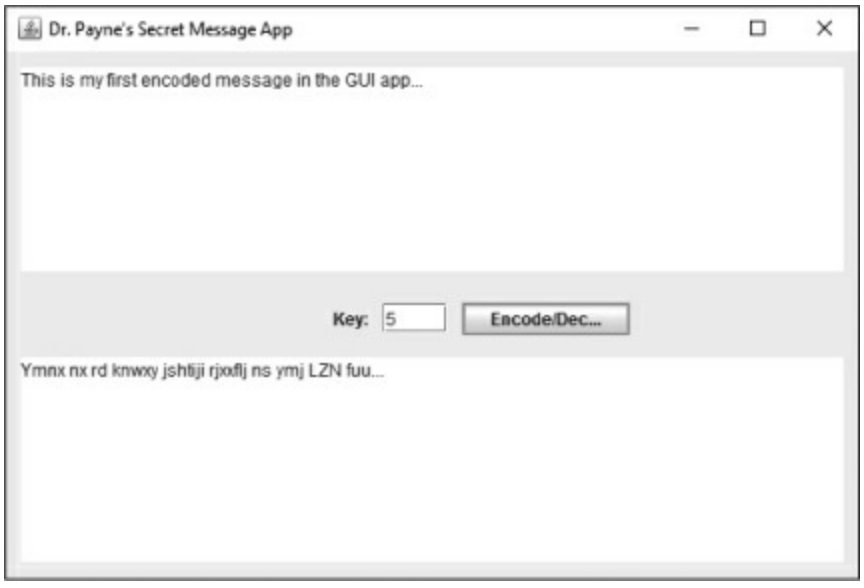

Dr. Payne's Secret Message App
This is my first encoded message in the GUI app...
Key:
5
Encode/Dec...
Ymnx nx rd knwxy jshtiji rjxoflj ns ymj LZN fuu...

Figure 7-7: The Secret Messages app is still a bit plain, but it can encode and decode messages!

The following is the full source code for *SecretMessagesGUI.java* to this point:

```
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
public class SecretMessagesGUI extends JFrame {
    private JTextField txtKey;
    private JTextArea txtIn;
    private JTextArea txtOut;
    public String encode( String message, int keyVal ) {
        String output = "";
        char key = (char) keyVal;
        for ( int x = 0; x < message.length(); x++ ) {
            char input = message.charAt(x);
            if (input >= 'A' && input <= 'Z')
            {
                input += key;
                if (input > 'Z')
                    input -= 26;
                if (input < 'A')
                    input += 26;
            }
            else if (input >= 'a' && input <= 'z')
            {
                input += key;
                if (input > 'z')
                    input -= 26;
                if (input < 'a')
                    input += 26;
            }
            else if (input >= '0' && input <= '9')
            {
                input += (keyVal % 10);
                if (input > '9')
                    input -= 10;
                if (input < '0')
                    input += 10;
            }
            output += input;
        }
        return output;
    }
    public SecretMessagesGUI() {
        setTitle("Dr. Payne's Secret Message App");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        getContentPane().setLayout(null);
```

```
        txtIn = new JTextArea();
        txtIn.setBounds(10, 11, 564, 140);
        getContentPane().add(txtIn);
        txtOut = new JTextArea();
        txtOut.setBounds(10, 210, 564, 140);
        getContentPane().add(txtOut);
        txtKey = new JTextField();
        txtKey.setBounds(258, 173, 44, 20);
        getContentPane().add(txtKey);
        JLabel lblKey = new JLabel("Key:");
        lblKey.setBounds(202, 176, 46, 14);
        getContentPane().add(lblKey);
        JButton btnEncodedecode = new JButton("Encode/Decode");
        btnEncodedecode.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                try {
                    String message = txtIn.getText();
                    int key = Integer.parseInt( txtKey.getText() );
                    String output = encode( message, key );
                    txtOut.setText( output );
                  } catch (Exception ex) {
                  }
          }
         });
         btnEncodedecode.setBounds(312, 172, 144, 23);
         getContentPane().add(btnEncodedecode);
    }
    public static void main(String[] args) {
        SecretMessagesGUI theApp = new SecretMessagesGUI();
        theApp.setSize(new java.awt.Dimension(600,400));
        theApp.setVisible(true);
    }
}
```

This first version of the app is a success, but it leaves a bit to be desired visually. We can make the GUI interface look more professional with just a few tweaks.

Improving the GUI

Let's switch back to the design view to correct the issues we noted in the last section. First, click the **Encode/Decode** button and make it a bit wider. Different computers can have other default fonts and sizes, so give the button a little extra room to be safe.

Now, let's make the font sizes in the two `JTextArea` fields larger and easier to read. We want to change the font properties of both text areas, so we can select both `txtIn` and `txtOut` at the same time by clicking `txtIn` and then using CTRL-

click (or ⌘-click) to also select `txtOut`.

With both text areas selected, go to the `font` property in the Properties pane and click the three dots next to the font size. The Font Chooser dialog will appear, as shown in Figure 7-8.

Figure 7-8: We can change the font properties for both JTextArea fields at the same time.

Choose a font size of 18 or greater and select a font that you like. Stick with a familiar font so that when you share the app with friends on different computers, the look and feel won't change too dramatically. I've selected Lucida Console, 18 point. Click **OK** to save the font selection.

Finally, let's change the background color of the content pane to personalize it. Select the content pane by clicking **getContentPane()** in the Components pane in the upper left. Then, click the three dots beside the `background` property in the Properties pane. In the Color Chooser dialog that pops up, select the **Named colors** tab, as shown in Figure 7-9. You'll see a broad selection of colors. Choose a color that you like and click **OK**.

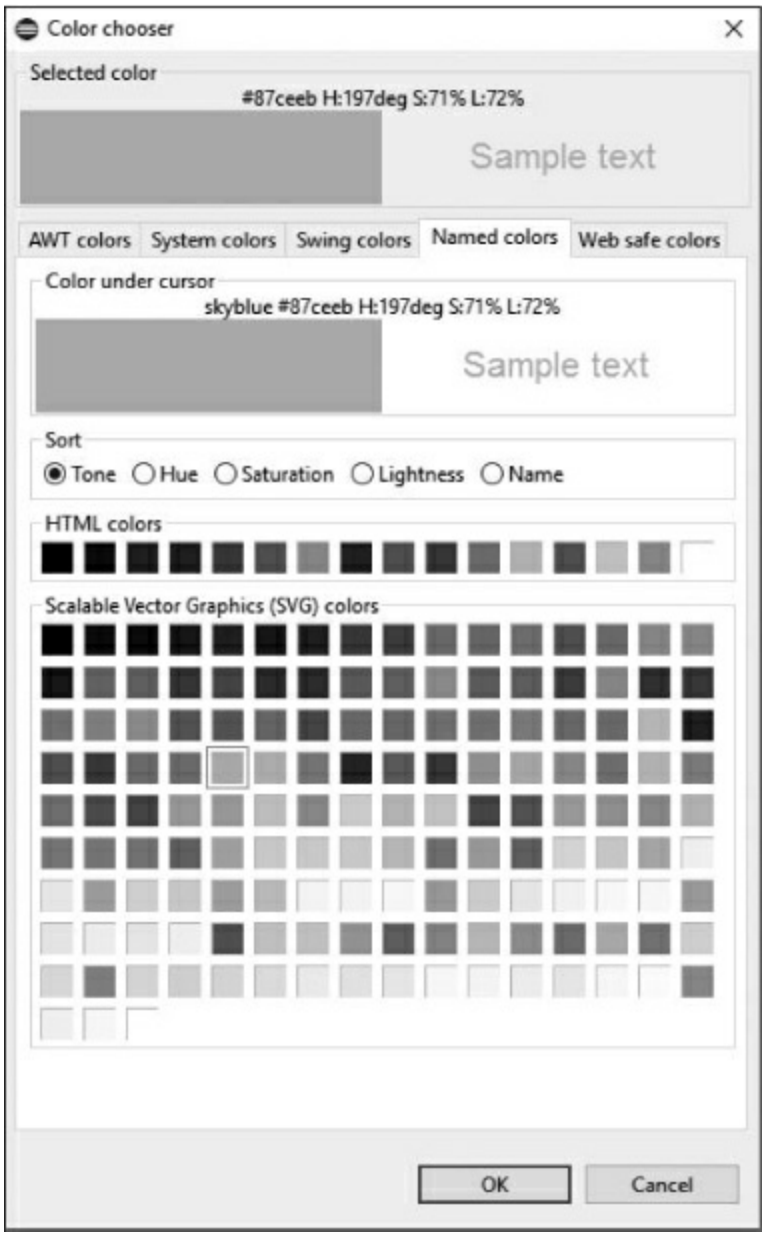


Figure 7-9: Change the background color of the content pane and button to make the app more stylish.

You can also change the background color of the Encode/Decode button. You may not see all your changes in the design preview, but if you run the app, you should see your new selections, as shown in Figure 7-10, along with any new background color you chose.

Figure 7-10: The GUI after you change the button size and font size.

There are a lot of other properties you can customize based on your preference. Explore the Properties pane for each component and adjust things until the app looks how you want it. Remember to stop and save your app so your changes don't get erased!

Setting Line Wrap and Word Wrap

So far, the messages I've used in my examples have been shorter than one line. But what happens if we type several lines of text?

Let's try typing or pasting a long sentence into the input text area. I'll use the first sentence of the Gettysburg Address, by US president Abraham Lincoln:

> Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

You can use this sentence or make up your own long sentence. Either way, as soon as you type it in, you'll notice a problem, as shown in Figure 7-11.

Dr. Payne's Secret Message App
Four score and seven years ago our fathers brought
Key: 13
Encode/Decode
Sbhe fpber naq frira lrnef ntb bhe snguref oebhtug

Figure 7-11: Longer messages in the app don't yet automatically wrap at the end of a line.

To correctly wrap longer lines of text by full words, you'll need to change *two* properties for each text area in the Properties pane. Select both `JTextArea` objects again in the design preview and then, under Properties, select the checkboxes next to `lineWrap` and `wrapStyleWord`. The first property, `lineWrap`, tells Java to wrap text longer than one line down to the next line. Setting the wrap style to `wrapStyleWord` tells Java to break the line at the end of a whole word, as we do in a word processor or as you see in this book.

Now save and run the app again. Paste your long sentence into the first text box, and you should notice an improvement. Now the text is correctly wrapped across multiple lines, with line breaks after the last full word that fits on each line. Enter a key and press the **Encode/Decode** button, and you'll see the same text wrapping in the output text area, as shown in Figure 7-12.

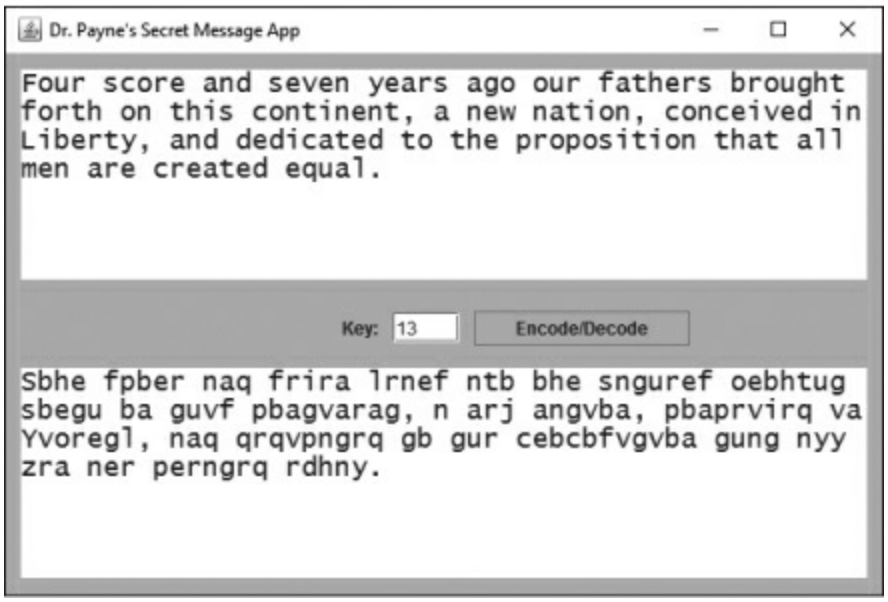

Dr. Payne's Secret Message App
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.
Key: 13
Encode/Decode
Sbhe fpber naq frira lrnef ntb bhe snguref oebhtug sbegu ba guvf pbagvarag, n arj angvba, pbaprvirq va Yvoregl, naq qrqvpngrq gb gur cebcbfvgvba gung nyy zra ner perngrq rdhny.

Figure 7-12: Changing both the lineWrap and wrapStyleWord properties to true results in a nice multiline interface, complete with word wrapping.

This version of the program is becoming more professional looking, but what happens if we enter bad input in the text field for the encryption key? Well, nothing yet. We added a `try` statement earlier, but we didn't complete the `catch` statement. Let's do that now. We'll also learn how to create pop-up message dialogs to alert the user to errors.

Handling Bad Input and User Errors: Part 2

Earlier in the chapter, we added a `try` block around the button event handler, but we left the body of the `catch` statement empty. It's time to fill that in with code.

Our `catch` statement will handle bad or missing user input in the `txtKey` text field. If the user forgets to enter a value for the encryption key or if the value they entered is not a number, one way of handling that is to pop up a message describing the problem, like the one shown in Figure 7-13.

Figure 7-13: A dialog pops up to alert the user to an error.

After the user closes the pop-up window, another helpful touch is to move the cursor to the `txtKey` text field and highlight or select any text there so the user can type a new value easily, without needing to click and select the value manually.

The `javax.swing` toolkit that we've been using for GUI components such as `JFrame` and `JTextField` also has classes that handle various kinds of pop-up windows. Some examples are *input dialogs*, which allow the user to provide text input; *confirm dialogs*, which allow the user to select Yes/No/OK or Cancel; and *message dialogs*, which provide information to the user in a pop-up window.

The error message we'd like to provide to the user is best displayed in a message dialog. The `javax.swing.JOptionPane` class has a method called `showMessageDialog()` that takes two parameters, which are the parent window and the message to display. If we were writing a larger application, we would use the parent window parameter to center the dialog over the main word processing window, for example. But for this app, we can use the keyword `null` to center the dialog over the desktop instead, as shown here:

```
JOptionPane.showMessageDialog(null,
      "Please enter a whole number value for the encryption key.");
```

When this line of code is called, it will pop up a message dialog reading "Please enter a whole number value for the encryption key."

To add this pop-up error message to the `catch` statement we created earlier,

go to the design view and double-click the **Encode/Decode** button. This is a quick way to have Eclipse take you directly to the source code for the button-click event handler that contains the `catch` statement.

After the error message is displayed, we'll also want to place the cursor in the `txtKey` field and select any text currently in the `txtKey` field, as we did for the Hi-Lo guessing game GUI app. We'll add the message dialog and the same `requestFocus()` and `selectAll()` statements we used in the Hi-Lo guessing game app inside the braces for the `catch` statement. The finished event handler code for the Encode/Decode button looks like the following:

```
        btnEncodedecode.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                try {
                    String message = txtIn.getText();
                    int key = Integer.parseInt( txtKey.getText() );
                    String output = encode( message, key );
                    txtOut.setText( output );
                } catch (Exception ex) {
                    JOptionPane.showMessageDialog(null,
"Please enter a whole number value for the encryption key.");
                    txtKey.requestFocus();
                    txtKey.selectAll();
                }
            }
        });
```

NOTE

When you add JOptionPane, if you don't use the content assist or if you see an error at JOptionPane, you may need to press ctrl-shift-O to import the javax.swing .JOptionPane class correctly at the top of your code.

Run the app now, either leaving the key field blank or entering nonnumeric text as the key. The message dialog will pop up, asking you to enter a valid key. It will also highlight the text in the key field so that it's easy to type a whole number when the dialog closes.

A first-time user might not know what to type into the key field, so adding a default value might also be a helpful UI tweak. Click the Design tab, select the `txtKey` text field, and enter a default key value into the `text` property in the Properties pane. I'll choose `3` as my default encryption key, but you can choose

any number. Let's also change the `horizontalAlignment` property of the `txtKey` field to **CENTER**. Feel free to change other properties to make `txtKey` look exactly like you want, including the font color and style.

Adding a Slider to the Secret Messages GUI

We'll add one more user interface improvement to the Secret Messages app—a numeric slider that will allow the user to quickly slide through key values and see how the message changes with each new key.

In design view, under Components in the Palette, select the **JSlider** component. Hover over the middle left of the design preview, beside the key prompt, and click to drop the `JSlider` into place as shown in Figure 7-14—you can adjust the position later if needed.

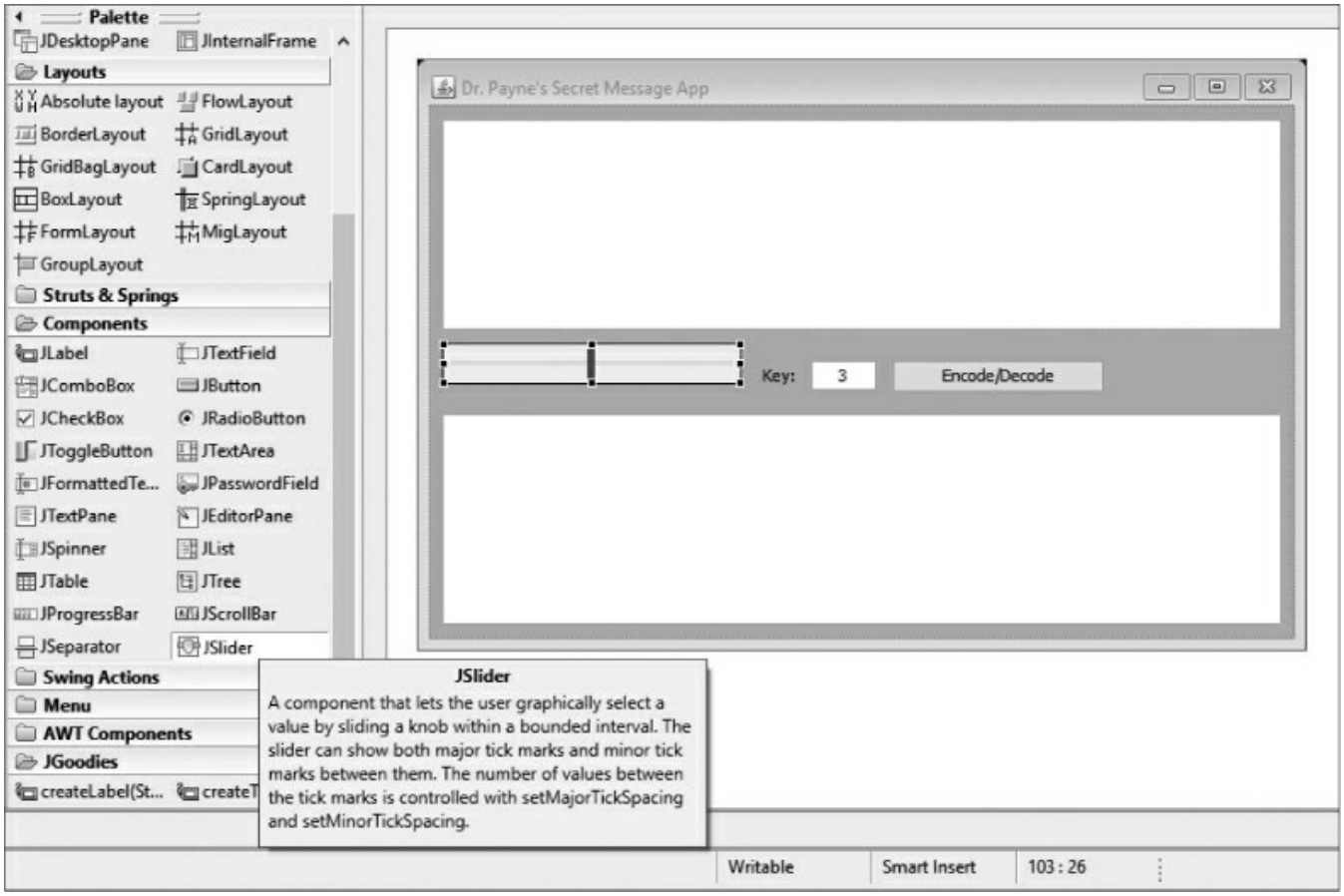


Figure 7-14: Add a JSlider to the graphical user interface for a quick, easy way to try different key values.

Now let's customize some properties of the `JSlider`. First, let's change its background color to match the rest of the app. With the new `JSlider` selected in

the design preview, click the three dots next to the `background` property in the Properties pane, go to **Named colors** in the Color Chooser dialog, and pick the same color you used for the rest of the GUI. Click **OK** to save the new background color for the slider.

Next, we'll add custom labels, tick marks, and a default value for the slider. First, turn on the `paintLabels` property by selecting the **true** checkbox beside it. Second, enter a `maximum` value of **`26`** and a `minimum` value of **`-26`** to make it easy to choose a key value in the correct range.

Next, for the `minorTickSpacing` property, enter a value of **`1`**—this will add small tick marks on the slider track to help the user see the range of possible values. Then, for the `majorTickSpacing` property, enter **`13`**—this will show a label every 13 values between `-26` and `26` on the slider. Now, turn on the `paintTicks` property by selecting the **true** checkbox next to it. The complete Properties pane with all the changes we've made is shown in Figure 7-15.

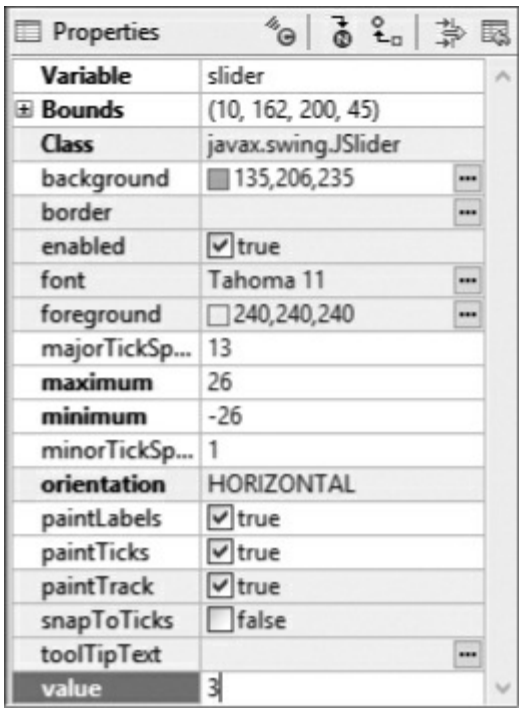


Figure 7-15: Customizing the properties of the slider

We're almost done! Change the default value of the slider to match your default key value in the key text field by modifying the `value` property. I selected `3` as my default key value, so I'll enter **`3`** in the `value` property for the slider. Finally, you'll need to make the slider a bit taller to show the labels, as shown in the finished slider in Figure 7-16.

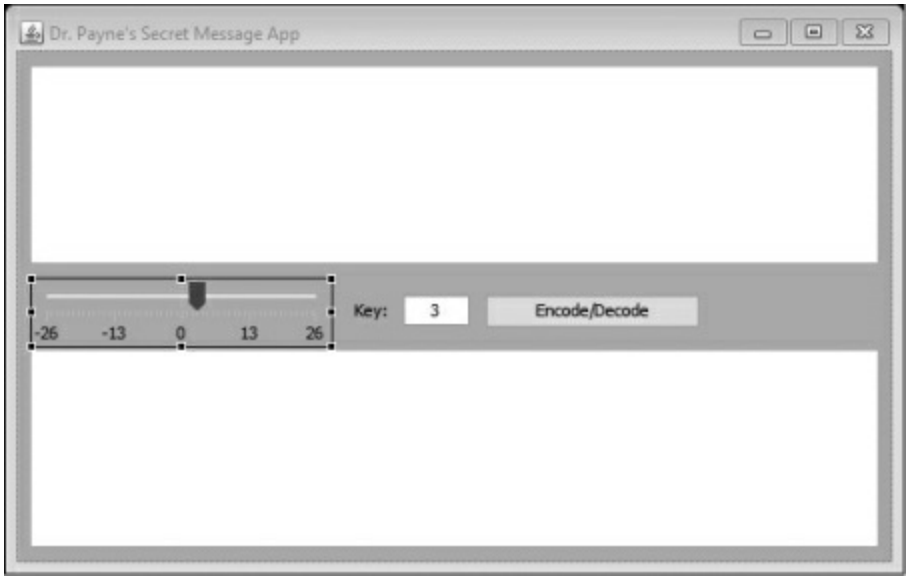


Figure 7-16: The Secret Messages app with the GUI slider in place

Now, we need to add the code to make the slider work, allowing the user to slide easily among various key values to encode and decode messages quickly.

Code Cracking with the Slider

We want the user to be able to change the encryption key value by clicking and dragging the slider we added in the previous section, so next we're going to add an event handler to listen for changes to the slider.

Right-click or CONTROL-click the slider in the design preview and go to **Add event handler ▸ change ▸ stateChanged**, as shown in Figure 7-17. The `stateChanged` event handler will work similarly to the `actionPerformed` event handler for the Encode/Decode button, but it will run anytime the user changes the slider's position to update the encryption key value.

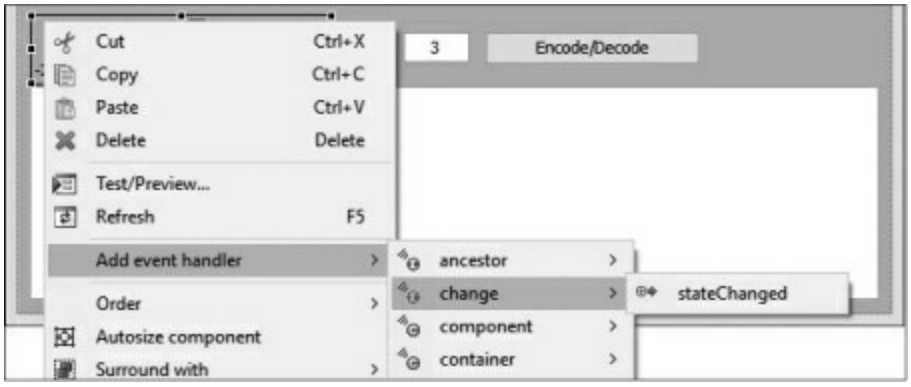


Figure 7-17: Adding an event handler to the slider that will detect when the user changes the slider's

value

When you click to add the `stateChanged` event handler to the slider, Eclipse will automatically generate another anonymous inner class, much like the one it created for the Encode/Decode button. But before we check out the anonymous inner class, we'll need to add the `JSlider` to the top of the class with the `JTextArea` declarations. To do that, we have to declare the `JSlider` as an instance variable. Add the last line shown here to the declarations at the top of the `SecretMessagesGUI` class:

```
public class SecretMessagesGUI extends JFrame {
    private JTextField txtKey;
    private JTextArea txtIn;
    private JTextArea txtOut;
    private JSlider slider;
```

Then, scroll back down to the slider code. Remember, you can switch back to design view and right-click to add the event handler again to go directly to the code anytime you get lost. Now, change the first line, removing the `JSlider` type declaration, as shown here:

```
        slider = new JSlider();
        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent arg0) {
```

The first action we'd like the slider to do is update the `txtKey` text field to show the slider's current position. We can get the slider's current position with the `getValue()` method, and we can set the text of the `txtKey` field with its `setText()` method. Adding those together, we can write the first line of the `stateChanged()` method:

```
        public void stateChanged(ChangeEvent arg0) {
            txtKey.setText( "" + slider.getValue() );
```

The `slider.getValue()` method returns an integer—the current position value of the slider—so we added an empty string to the number to turn it into a string or text value. This will change the `txtKey` text field to show the slider's value. But it doesn't automatically re-encode the message using the new value yet. If you run the app now, you still need to click the Encode/Decode button to change the message.

Let's modify the code in the `stateChanged()` method so that the slider updates

the message like the Encode/Decode button would. We can copy the code from inside the `try` statement for the Encode/Decode button and paste it after the first line of the `stateChanged()` method for the slider, making only one change:

```
slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent arg0) {
        txtKey.setText( "" + slider.getValue() );
        ❶ String message = txtIn.getText();
        ❷ int key = slider.getValue();
        ❸ String output = encode( message, key );
        ❹ txtOut.setText( output );
    }
});
```

The lines at ❶, ❸, and ❹ are copied directly from the Encode/Decode button's event handler, but we've made a small change at ❷. Instead of parsing the value out of the text field, we can access the slider's value directly as an integer using the `getValue()` method. This means we don't need a `try-catch` statement, because we're no longer relying on the user to supply a hand-typed value. This is more convenient for the user and more secure for the programmer, as using the GUI slider removes one potential error while also making the interface easier to use.

Save the changes and run the application, and you'll see that you can type in a message and then change the position of the slider to try different encryption key values. To test the ability to decode, copy an encoded message from the bottom text area using CTRL-C (or ⌘-C) and paste it into the top text area using CTRL-V (or ⌘-V). Then slide the slider to the left or right until you see the original message again.

You'll even be able to "crack" the Caesar cipher encryption for messages that you don't have the key to by sliding the slider slowly until you can read the decoded message in the bottom text area. Give it a try! Type the following message into the top text area of your app:

```
Epdetyr esp lmtwtej ez mcplv esp Nlpdlc ntaspc htes xj Dpncpe Xpddlrp laa...
```

Then, slide the slider to the far left and slowly slide it to the right until it reveals the secret message. Can you tell what secret key was used to encode the message? See Figure 7-18 for a hint.

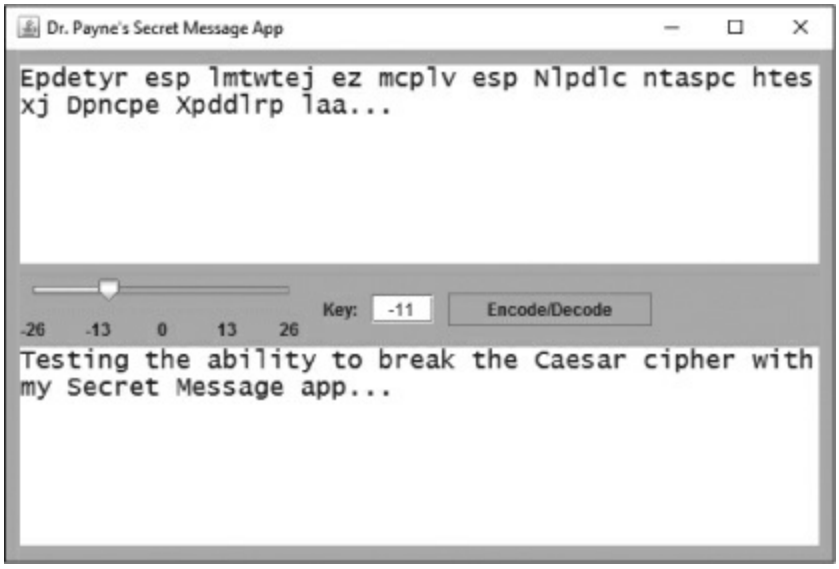

Dr. Payne's Secret Message App
Epdetyr esp lmtwtej ez mcplv esp Nlpdlc ntaspc htes xj Dpncpe Xpddlrp laa...
Key: -11
Encode/Decode
-26 -13 0 13 26
Testing the ability to break the Caesar cipher with my Secret Message app...

Figure 7-18: You can use the Secret Messages app to break Caesar cipher messages by sliding the slider left and right until a plaintext message appears!

Moving the slider to -11 results in a plaintext English message, so the secret key must have been 11. You may also notice that 15 will break the message (because 15 = 26 – 11). You can always find a couple of values that work to break messages in the basic Latin alphabet. Of course, you can send messages in other languages as well, as shown in Figure 7-19.

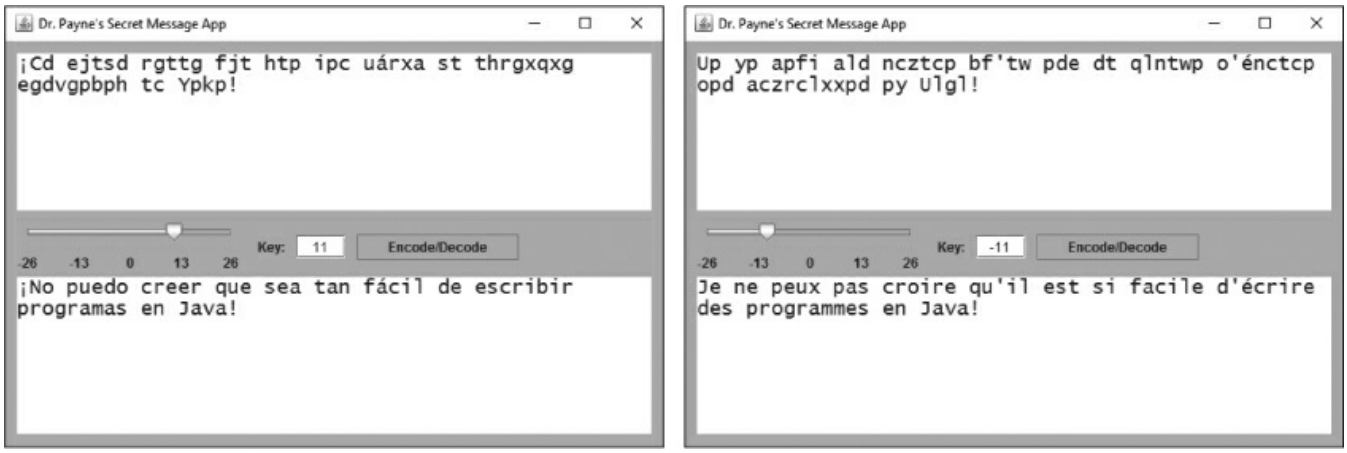

Dr. Payne's Secret Message App
¡Cd ejtsd rgttg fjt htp ipc uárxa st thrgxqxg egdvgpbph tc Ypkp!
Key: 11
Encode/Decode
-26 -13 0 13 26
¡No puedo creer que sea tan fácil de escribir programas en Java!
Dr. Payne's Secret Message App
Up yp apfi ald ncztcp bf'tw pde dt qlntwp o'énctcp opd aczrclxxpd py Ulgl!
Key: -11
Encode/Decode
-26 -13 0 13 26
Je ne peux pas croire qu'il est si facile d'écrire des programmes en Java!

Figure 7-19: Any languages that use characters in the basic Latin alphabet can be used with this version of the Secret Messages app to send and receive encoded messages; Spanish (left) and French (right) are shown here.

You can copy and paste encoded messages back and forth with your friends using email, Twitter, and Facebook—you can even send encoded text messages, although that would be easier to do from your mobile device. We'll see how to turn this Secret Messages desktop app into a mobile app in Chapter 8.

For now, in order to share secret messages back and forth, it would be nice to be able to share your Secret Messages app with your friends easily. We'll learn to do that in the following section.

Bonus: Sharing Your App as a Runnable JAR File

For it to be really useful, you need to be able to share the Secret Messages app with your friends—possibly even with friends who don't know how to code in Java and don't have Eclipse installed on their computer.

The great news is that Eclipse and Java make it easy to export an application as a runnable file and share it with any of the millions of computers worldwide that run Java. Your friends don't even have to download the Java JDK developer's edition; all they need is the JRE, which comes installed on most computers. Eclipse can export an application in a runnable Java archive (JAR) file that you can share by email, USB flash drive, or some other means. All your friends have to do is double-click the JAR file, and they'll be able to run your app!

To export a runnable JAR file from Eclipse, go to **File ▸ Export**. Then expand the *Java* folder and click **Runnable JAR file**, as shown in Figure 7-20.

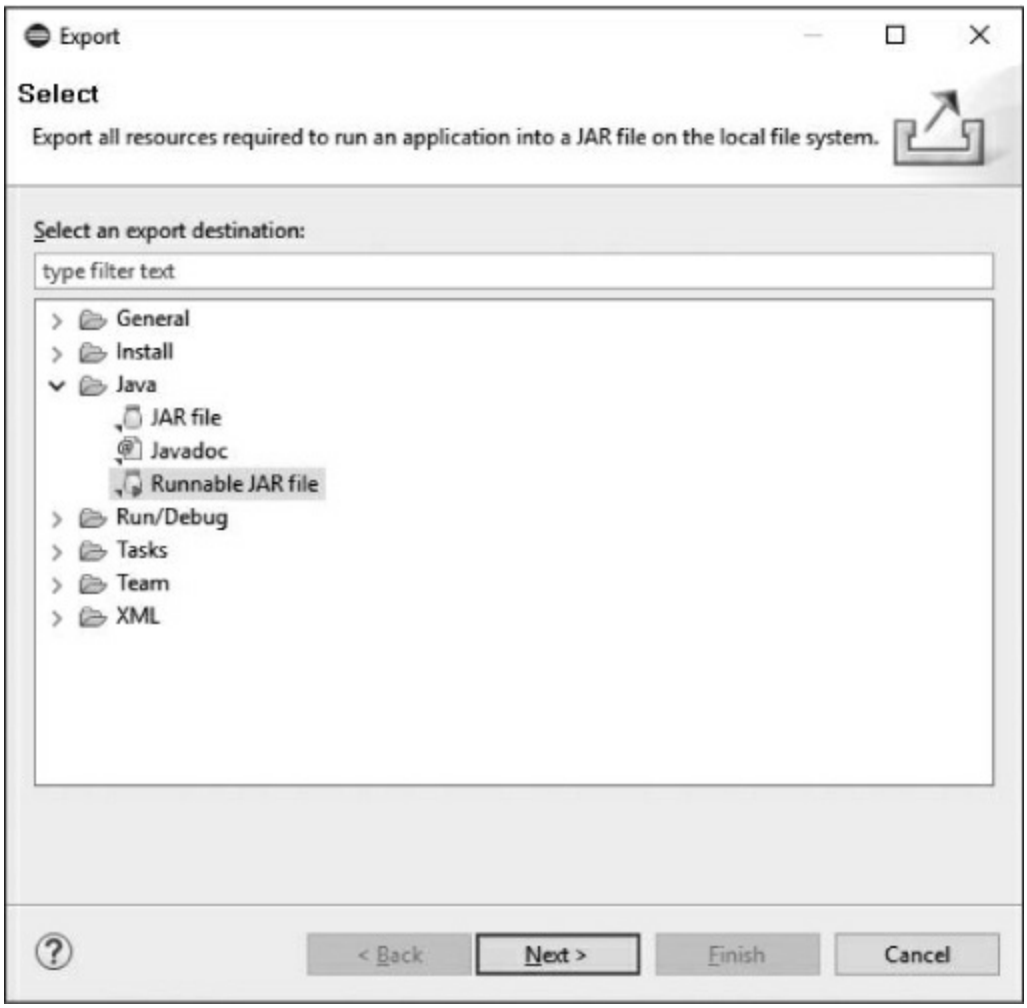


Figure 7-20: Share your application with others by exporting a runnable JAR file.

Click **Next**, and Eclipse will ask you for a launch configuration and an export destination. The launch configuration means which class or app you'd like to run and from which project. Under **Launch configuration**, click the drop-down list and choose **SecretMessagesGUI – SecretMessagesGUI**. This means you want to run the *SecretMessagesGUI.class* file in the *SecretMessagesGUI* project.

NOTE

You have to compile and run your application at least once to have a launch configuration for that app.

The export destination is just the location and filename you want to use for your executable app. Under **Export destination**, click **Browse** and then select the destination folder you'd like to save your finished app in, such as your

Desktop folder, for example. Give your program file a name, like *Secret Messages.jar*. You can use spaces (or other special characters) to name your file exactly how you'd like people to see it when they open it. When you've made your selections, as shown in Figure 7-21, click **Save** and then **Finish**.

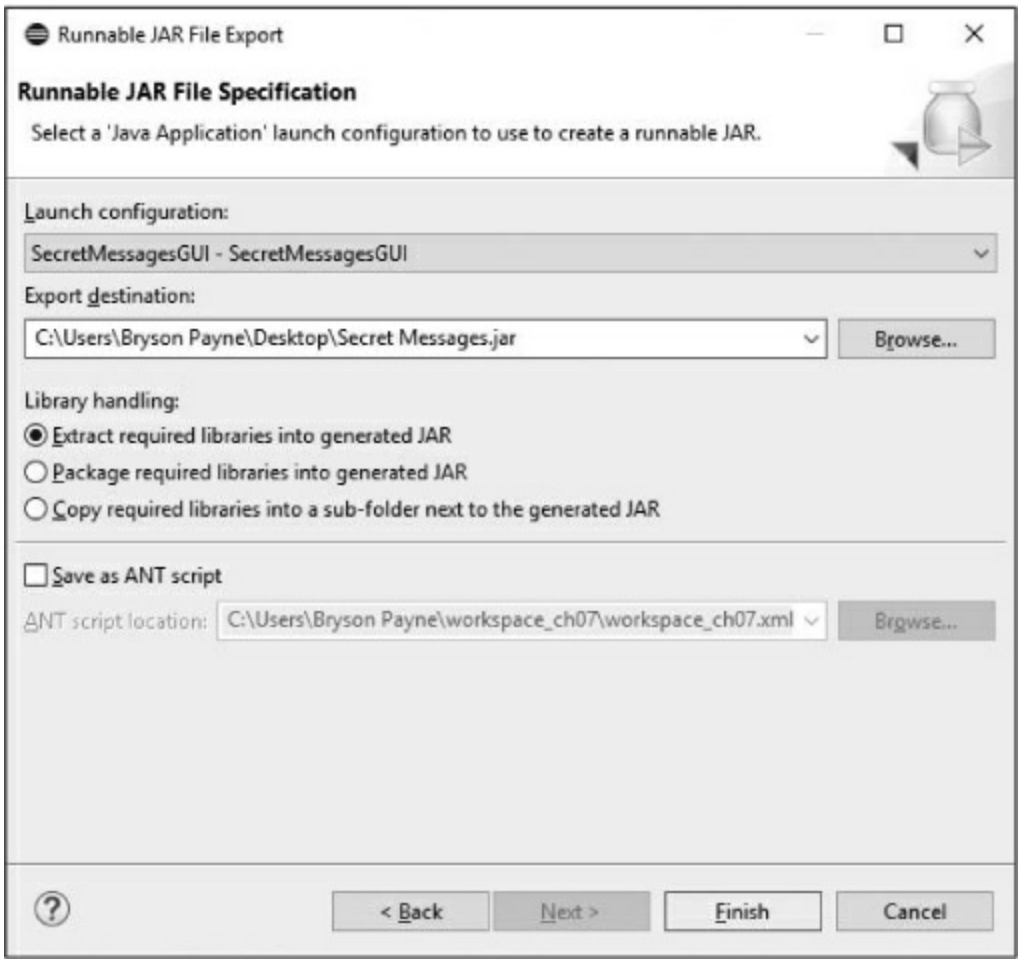


Figure 7-21: Choose a location and name for your runnable JAR file.

You may see a warning after clicking Finish, but you can ignore it and click **OK**. Go to wherever you saved your JAR file, and you should see a Java icon with your filename, as shown in Figure 7-22. Now you should be able to run the program and send and receive coded messages.

Figure 7-22: The desktop icon for the executable JAR file, on Windows (left) and on macOS (right)

NOTE

On newer versions of macOS, you'll need to hold down CONTROL when you click the app the first time, then select ***Open****, and finally click* ***Open*** *on the dialog that pops up to allow the app to run.*

Some email programs may block files ending in *.jar* as a safety precaution, but you can upload the file to your favorite file-sharing program (Dropbox, Google Drive, or SkyDrive, for example) and share the file from there. Once your friend has the file, all they have to do is run it, and you can share scrambled messages with each other.

NOTE

You can also export a runnable JAR file for the GUI guessing game app from Chapter 3. Just follow these same steps for the GuessingGame.java *file in the* GuessingGame *project.*

What You Learned

You reused code from the text-based Secret Messages app in this app, and you substantially expanded your understanding of GUI interface design and programming. Here are some of the skills covered in this chapter:

- Naming GUI components and variables consistently for code readability and reusability
- Reusing algorithms, such as the `encode()` method that we copied from the text-based version of this app
- Declaring methods that accept parameters and return information
- Writing more advanced event handlers for a wider variety of GUI components, including buttons, sliders, and text fields
- Working with properties of GUI components in design view to customize the UI further
- Setting line wrap and word wrap for `JTextArea` objects

- Popping up `javax.swing.JOptionPane` dialogs, including message dialogs
- Adding and working with a `JSlider` GUI slider
- Changing a text field's text using a `stateChanged()` event handler on a slider
- Exporting a runnable JAR file to share your app with friends

Programming Challenges

Try these programming challenges to review and practice what you've learned and to expand your programming skills by trying something new. Visit the book's website at *https://www.nostarch.com/learnjava/* to download sample solutions.

#1: Movin' On Up!

The Secret Messages app is great for sending simple, encoded messages to your friends, but it's also fun to use all by yourself. One task that gets repetitive from time to time is copying encoded messages from the output text area and pasting them into the input text area. Your first programming challenge is to create a Move Up ^ button, which moves the encoded message up and automatically decodes it!

Place the Move Up ^ button next to the Encode/Decode button. Then, add an event handler that will get the text from the `txtOut` text area and set it as the text of the `txtIn` text area. As a bonus, have the event handler for the Move Up ^ button change the slider to the opposite of its value. You can also change the background of the button to match the rest of your app. A sample solution to this challenge is shown in Figure 7-23.

Figure 7-23: Adding a Move Up ^ button to swap the message and slider value to automatically decode your message

#2: Scrolling Away!

Another improvement we could make to the app would be to handle even longer messages. In this case, we should add a scroll bar along the side and scroll down automatically whenever the user enters a message too long for the input text area. The `JTextArea` won't add a scroll bar by itself, but Eclipse makes adding a scroll bar to any text area quick and easy!

In the design view, right-click either of the `JTextArea` boxes and select **Surround with ▸ javax.swing.JScrollPane**, as shown in Figure 7-24.

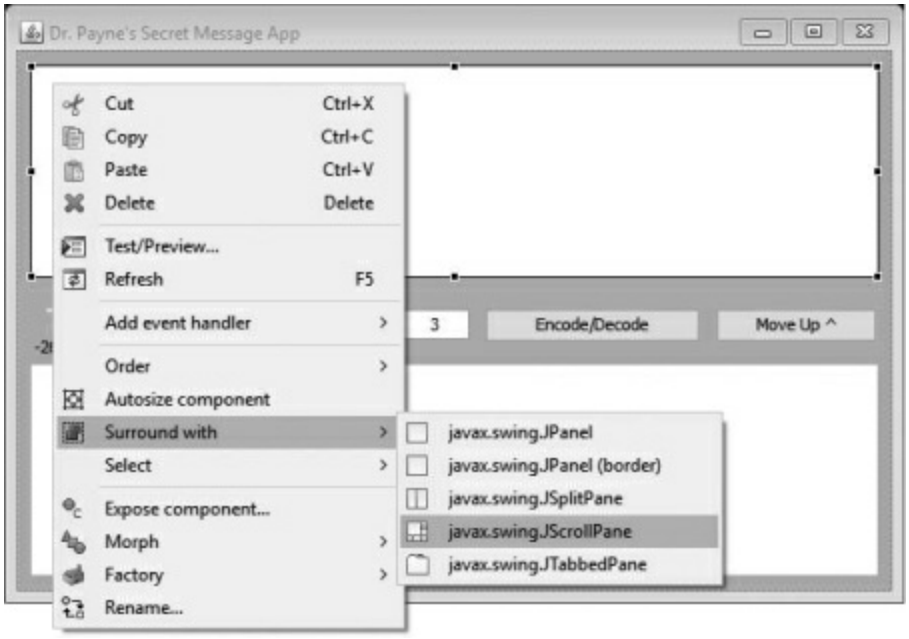


Figure 7-24: Adding an automatic scroll bar to a JTextArea

You'll need to do this for the other `JTextArea` as well. Run the app when you're finished, typing a long selection of text into the input text box to see the scroll bar appear automatically. In Figure 7-25, I've pasted the entire US Constitution into the input text area and encoded it.

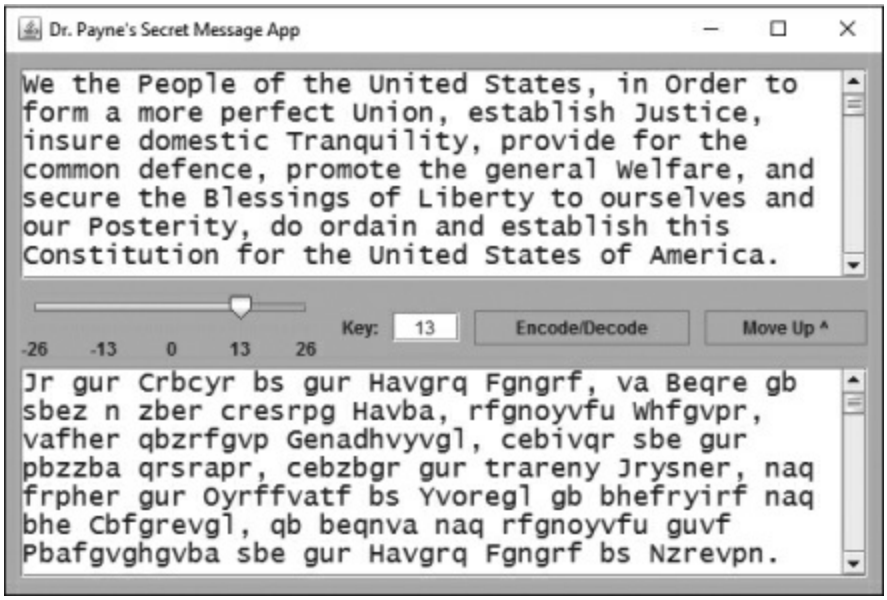

Dr. Payne's Secret Message App
We the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America.
Key: 13
Encode/Decode
Move Up ^
-26 -13 0 13 26
Jr gur Crbcyr bs gur Havgrq Fgngrf, va Beqre gb sbez n zber cresrpg Havba, rfgnoyvfu Whfgvpr, vafher qbzrfgvp Genadhvyvgl, cebivqr sbe gur pbzzba qrsrapr, cebzbgr gur trareny Jrysner, naq frpher gur Oyrffvatf bs Yvoregl gb bhefryirf naq bhe Cbfgrevgl, qb beqnva naq rfgnoyvfu guvf Pbafgvghgvba sbe gur Havgrq Fgngrf bs Nzrevpn.

Figure 7-25: By adding a JScrollPane around each JTextArea, you can encode text as long as the US Constitution or even longer!

Only part of the text is visible in Figure 7-25, but now there is a scroll bar along the right side of both text areas. The scroll bar lets us know that there's

more text than will fit into the text area, and all we have to do is scroll down to see it.

To customize the app even more, explore the Properties pane for each of the GUI components we've added and change various property values such as background colors, fonts, and so on. Make the app your own and share it with friends to show off what you've created!

#3: Changing the Text to Change the Slider

For this challenge, let's make one last tweak to the user interface. When the user slides the slider, the key value shown in the text field changes. But what if we want the slider to move whenever the value in the text field changes?

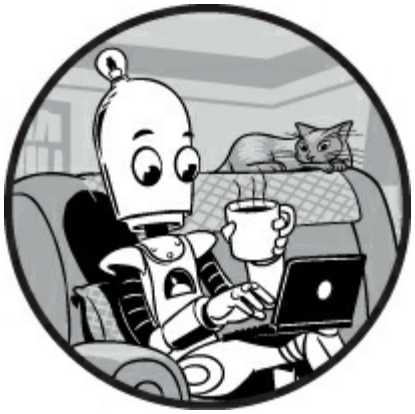
HINT

You'll want to add an event handler to the txtKey text field. Right-click the txtKey JTextField and then go to ***Add event handler ▸ key ▸ keyReleased****. This will create an event handler that listens for keystrokes in the txtKey text field.*

To accomplish this, you'll need to write code for the event handler that will get the integer value from the text field and set the slider to that value. Remember to use a `try-catch` block when dealing with user-supplied text. Good luck!

8

MAKE SECRET MESSAGES A PHONE APP TO SHARE WITH FRIENDS!



In this chapter, we'll complete our Secret Messages app experience by building a mobile app version that can send secret messages by text or email and can even post to social media.

The app will look similar to the GUI version in Chapter 7, with labels, text fields, and the ability to slide through different key values to quickly encode and decode messages. You'll also be able to copy and paste messages from the app into emails and text messages, but by the end of the chapter, you'll be able to send an email or text directly from the Secret Messages app itself! Figure 8-1 shows the app in action on a real Android device.

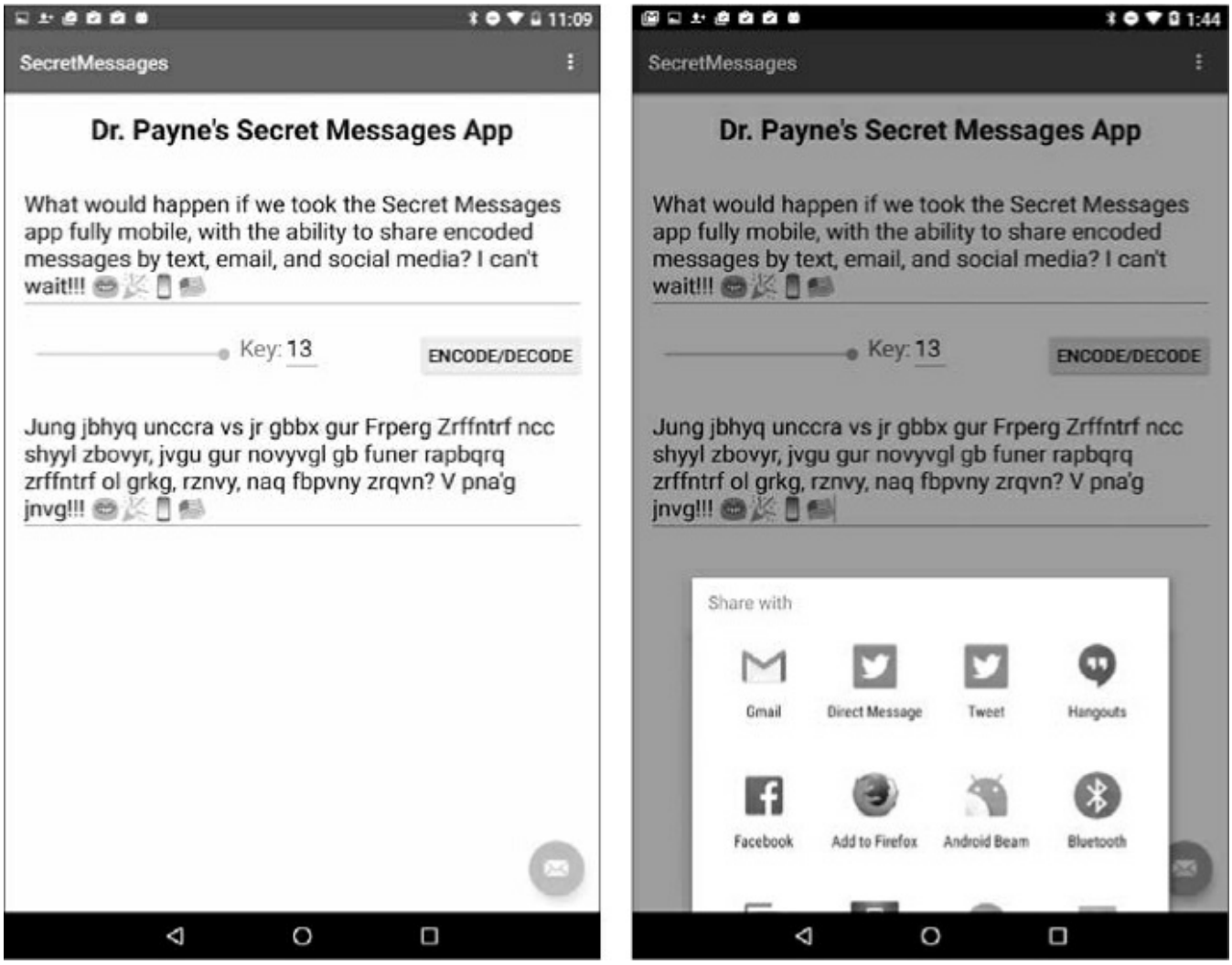

11:09
SecretMessages
Dr. Payne's Secret Messages App
What would happen if we took the Secret Messages app fully mobile, with the ability to share encoded messages by text, email, and social media? I can't wait!!!
Key: 13
ENCODE/DECODE
Jung jbhyq unccra vs jr gbbx gur Frperg Zrffntrf ncc shyyl zbovyr, jvgu gur novyvgl gb funer rapbqrq zrffntrf ol grkg, rznvy, naq fbpvny zrqvn? V pna'g jnvg!!!
1:44
SecretMessages
Dr. Payne's Secret Messages App
What would happen if we took the Secret Messages app fully mobile, with the ability to share encoded messages by text, email, and social media? I can't wait!!!
Key: 13
ENCODE/DECODE
Jung jbhyq unccra vs jr gbbx gur Frperg Zrffntrf ncc shyyl zbovyr, jvgu gur novyvgl gb funer rapbqrq zrffntrf ol grkg, rznvy, naq fbpvny zrqvn? V pna'g jnvg!!!
Share with
Gmail
Direct Message
Tweet
Hangouts
Facebook
Add to Firefox
Android Beam
Bluetooth

Figure 8-1: The Secret Messages encoder/decoder app (left) can send secret text and email messages directly from inside the app, at just the touch of a button (right).

We'll reuse the working code from the desktop app by keeping the names of the GUI components in the Android version and by taking advantage of Java's consistent language and structure across multiple platforms.

Setting Up the Mobile GUI

Let's begin building the Secret Messages app with Android Studio. Android Studio may ask you to perform updates as you open the program, which is usually a good idea. Updating carries a bit of risk, as it can change the look and feel of the program by moving icons or menu items, but the updates also include important new security fixes and the latest Android features.

When you first open Android Studio, you may see the most recent project

you were working on—in our case, the Hi-Lo guessing game app. You can close it by selecting **File ▸ Close Project**.

To begin building the new Secret Messages app, either select **File ▸ New Project** from inside Android Studio or choose **Start a new Android Studio project** from the Welcome screen.

Name the new project **`SecretMessages`**, save it in your preferred location, and click **Next**.

On the Target Android Devices screen, choose **API 16: Android 4.1 (Jelly Bean)** for the **Phone and Tablet Minimum SDK**, as we did for the Hi-Lo guessing game app, and then click **Next**. Choose **Basic Activity** on the Add an Activity screen and click **Next**. On the Customize the Activity screen, keep all the default names and click **Finish**.

It may take a moment for the new project to configure and open. When the new project opens, double-click the *content_main.xml* file under *app ▸ res ▸ layout* in the Project Explorer on the left. You may need to close a pop-up message about the floating action button (we'll use that later).

Your screen should look similar to Figure 8-2. (You may have to click the Design tab to see the GUI preview.)

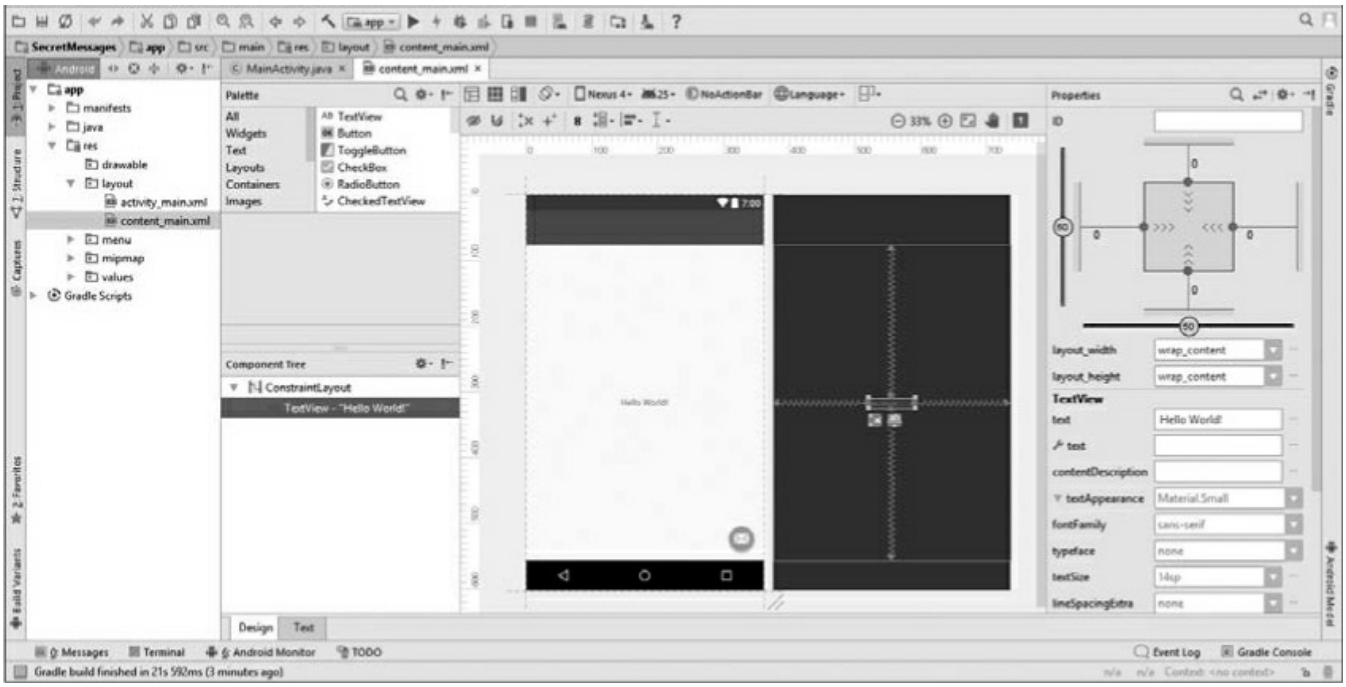


Figure 8-2: The starting view of the app in Android Studio

Designing the Mobile GUI

Now that we have a new project set up, let's start designing the layout of our app. First, delete the "Hello World!" text shown in the design preview by clicking to select it and then pressing the DELETE key. Then, as we did in the previous app, let's add a RelativeLayout to hold our GUI elements: in the Palette pane, under Layouts, select **RelativeLayout** and drag it onto the Preview pane or onto the ConstraintLayout under the Component Tree.

Next, let's add a title to the top of the app by selecting the **TextView** widget under Widgets in the Palette. Place the `TextView` widget at the top center of the screen, change the text to ***`Your Name`*`'s Secret Messages App`**, and then change the `textAppearance` property in the Properties pane to **Material.Large**, as I've done in Figure 8-3.

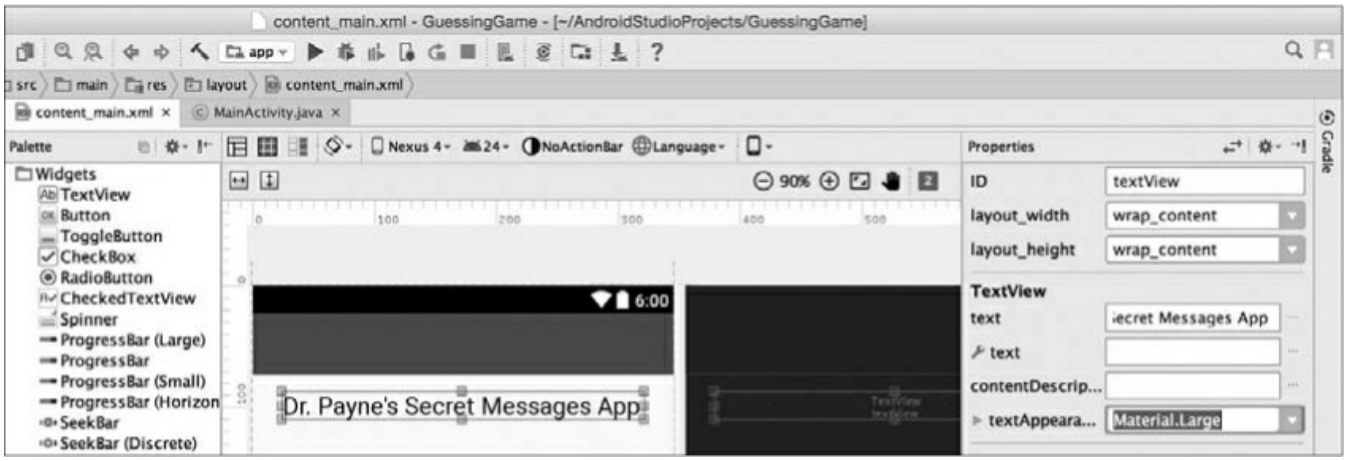


Figure 8-3: Add a Large text title to the top center of your app, customizing it with your name.

In the Properties pane, expand **textAppearance** to find **textStyle** and click the **bold** checkbox to make the title text stand out more.

Now, let's place the `EditText` text fields for the user's input message. In the Palette, under Text, select **Multiline Text**. Zoom in on the design preview and place the text field about 30 dp below the title. Change the text to **`Secret messages \n are so cool, \n aren't they?`** as shown in Figure 8-4. The `\n` escape sequence will add *newline characters*; these characters don't show up onscreen, but they represent the new line you get by pressing ENTER on the keyboard. Putting the text across multiple lines shows the user they can enter a multiline message as well.

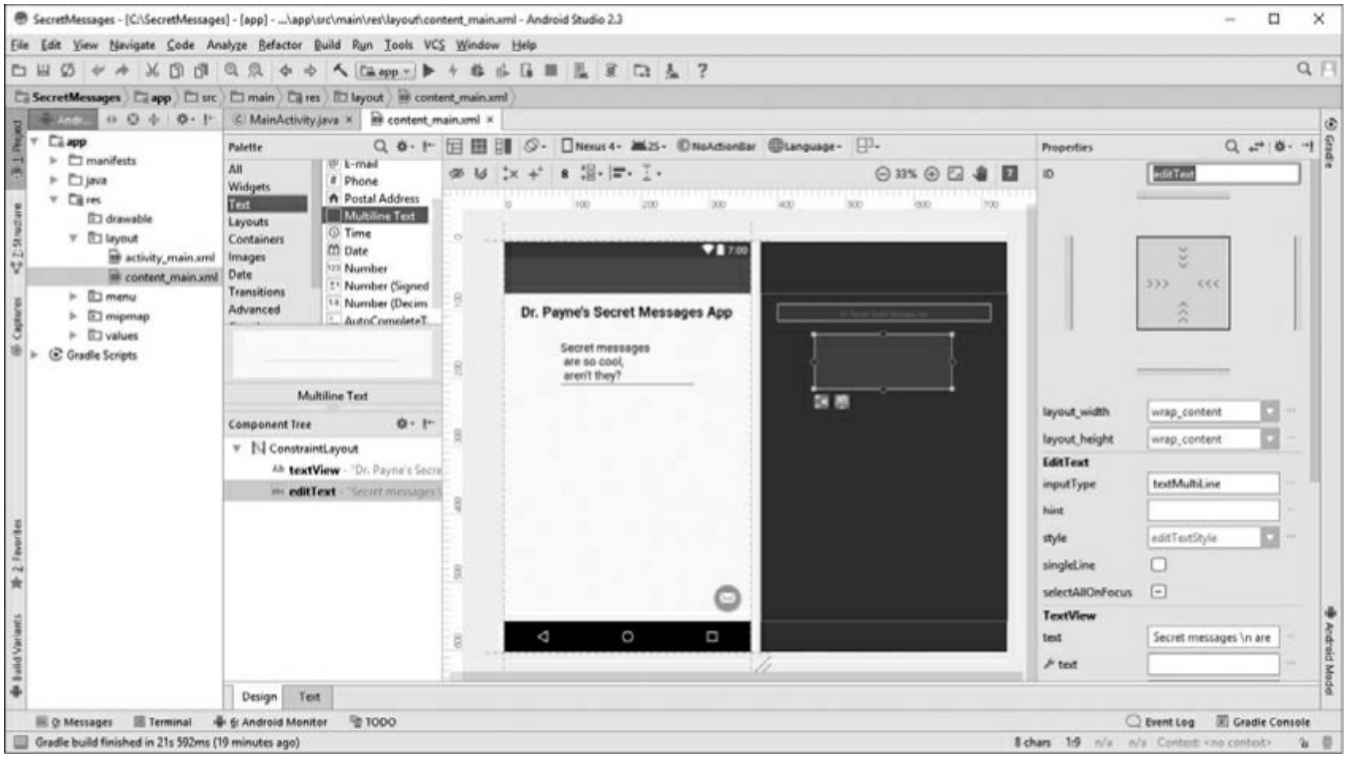


Figure 8-4: Add a Multiline Text field for the user's input message, using the \n escape sequence to span multiple lines of text.

You can also change the number of lines of text that a user can see at any one time. Under the expanded Properties pane for the Multiline Text, you can change the `lines` property to `4`, for example. To be able to see the multiline message across all devices, select the **textMultiLine** checkbox in the `inputType` property's drop-down list.

Finally, let's change the input text field's `id` property to `txtIn`, both to help us use it in the code later and to maintain naming consistency with the GUI desktop version of the app. Now drag the left and right borders of the text field to stretch it to the full width of the app, as shown in Figure 8-5.

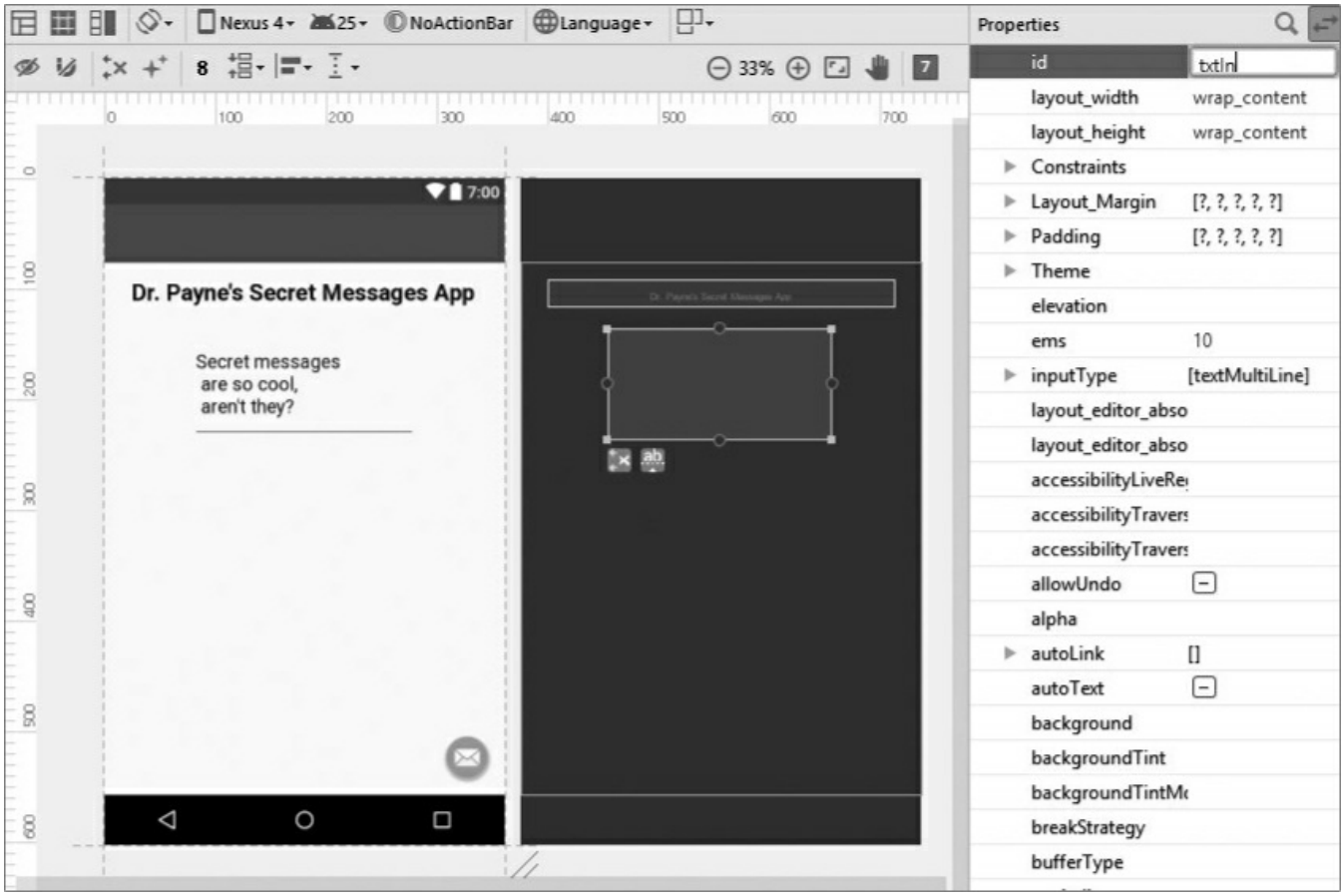


Figure 8-5: Change the id of the input text field to txtIn so that we can code with it later.

We're going to add four components to the middle of the Secret Messages app: a `SeekBar` for the user to swipe a bar to find a key, a label, a text field to display the secret key, and an Encode/Decode button. The `SeekBar` will function similarly to the `JSlider` in the desktop version of the app. Don't worry if all your components don't line up perfectly when you first place them—we'll be able to fix that later.

First, find the `SeekBar` in the Widgets section of the Palette pane. Place it a comfortable margin, 30 dp or so, below the input text field along the left margin, as shown in Figure 8-6. You can set the top margin manually in the Properties pane by going to **View all properties** (the left and right double-arrows at the top of the Properties pane). Under **Layout_Margin**, enter **`30dp`** for the **layout_marginTop**.

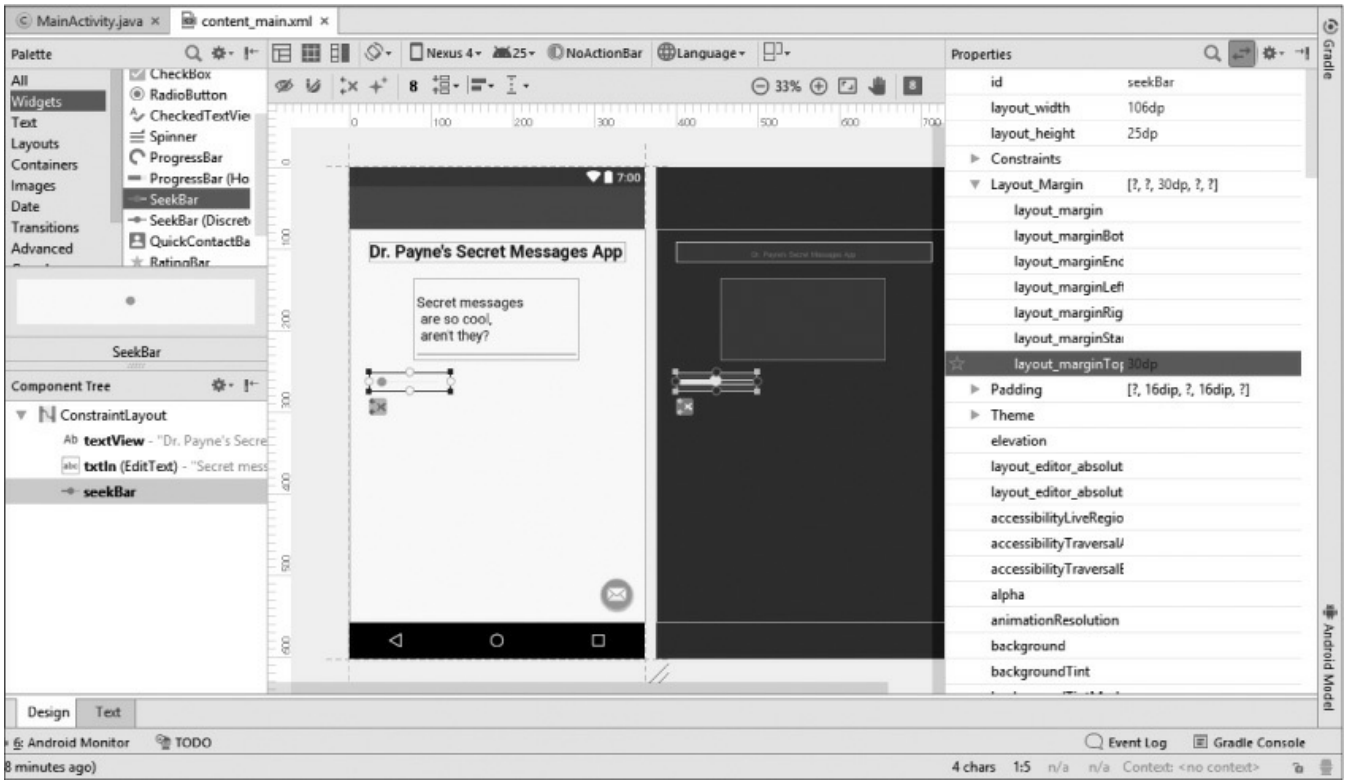


Figure 8-6: The SeekBar will function like the slider in the GUI desktop version of the app, allowing the user to slide among different key values.

Next, add a `TextView` widget beside the `SeekBar` to serve as a label for the text field of the user's secret key value. Change the `textAppearance` property to **Material.Medium** and the `text` property to **Key:** in the Properties pane.

Then, select a **Number** text field for the user's secret key value and center it horizontally, with a text property of `13` or another default key value of your choice. Modify the width of the text field to fit the number comfortably by changing its `width` property (under **View all properties**) to approximately **`40dp`**, and then change the `id` property to **`txtKey`**.

With the `txtKey` text field in place, you should be able to drag the `Key:` label over to be directly beside it. You should also be able to stretch the `SeekBar` to fill the gap, as shown in Figure 8-7.

Complete the middle row of the layout by adding the Encode/Decode button. Choose **Button** under Widgets in the Palette and insert the button along the right margin beside the text field. Change the `text` property to **`Encode/Decode`**.

Finally, create a text field for the output message by copying the `txtIn` text field and pasting it below the middle row of widgets. Delete the text in the text field, stretch it to the width of the app window, and rename the `id` property to **`txtOut`**.

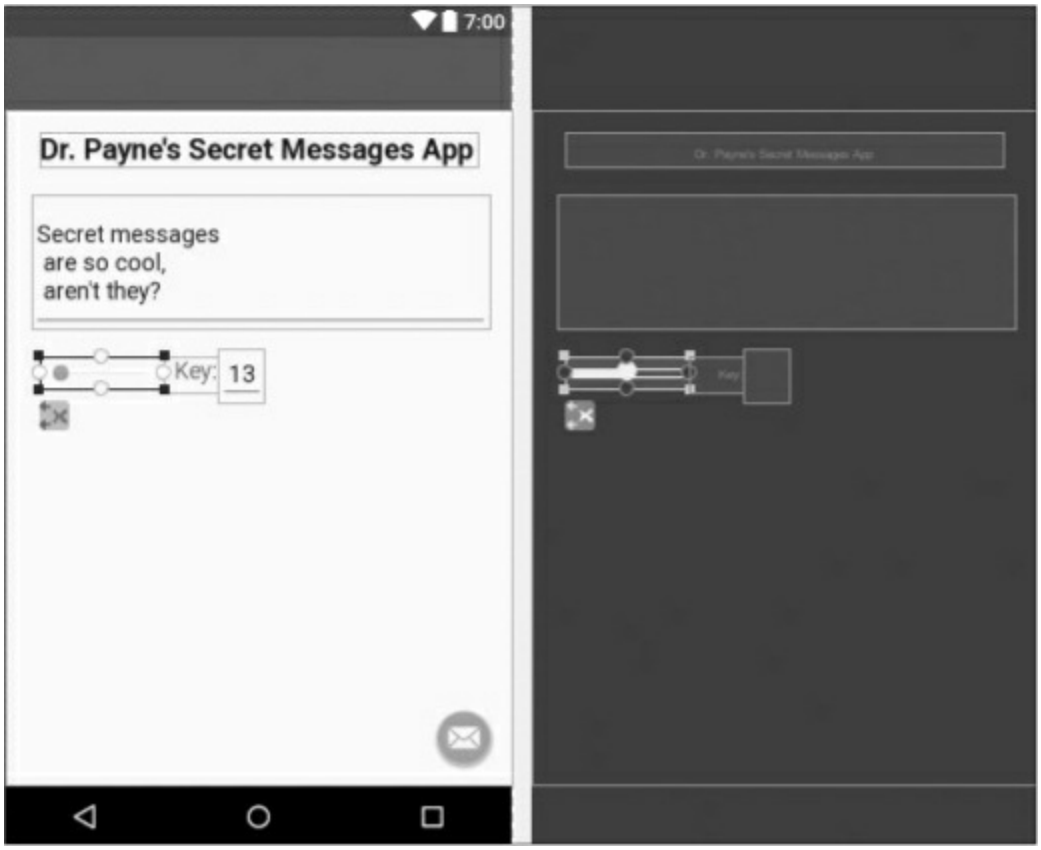


Figure 8-7: After placing the txtKey text field, adjust the placement of the Key: label and stretch the SeekBar.

That's it—the finished GUI layout is shown in Figure 8-8.

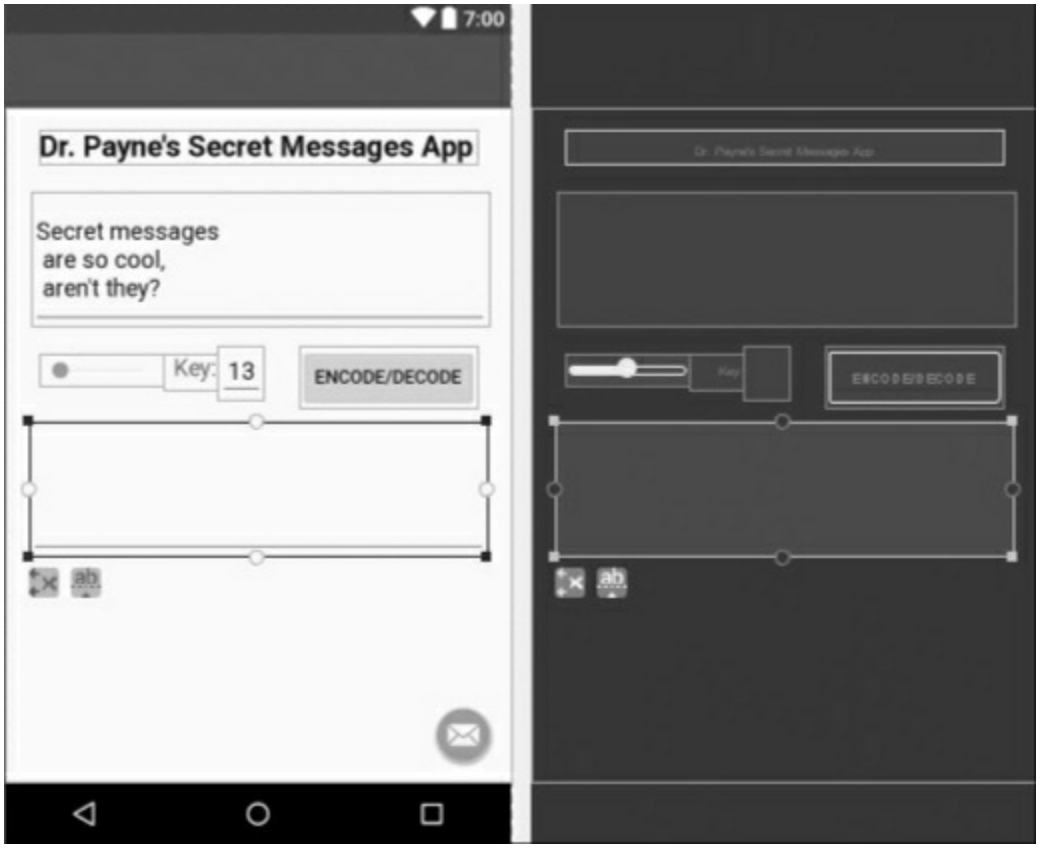


Figure 8-8: The finished layout for the Secret Messages mobile app

Before moving on, make sure your input text field has an `id` of `txtIn`, the output text is `txtOut`, and the middle text field for the key is `txtKey`. We'll need all of these elements to be correctly named in order for the code to work in the next section.

Wiring the GUI to the Java Code

It's time to connect the GUI layout to the Java source code to begin programming the behavior of the Secret Messages mobile app. Switch to the *MainActivity.java* file by clicking the tab at the upper left of the main content window.

Add the following five lines directly to the `public class MainActivity` declaration:

```
public class MainActivity extends AppCompatActivity {
    EditText txtIn;
    EditText txtKey;
    EditText txtOut;
    SeekBar sb;
```

```
    Button btn;
```

You may need to press ALT-ENTER or OPTION-ENTER after each line to import the widgets into the Java code.

These five variables point to the GUI components in the visual layout for the app. We can connect these variable names to the actual widgets by adding five lines of code to the `onCreate()` method, right below the `setSupportActionBar(toolbar);` statement:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    txtIn = (EditText)findViewById(R.id.txtIn);
    txtKey = (EditText)findViewById(R.id.txtKey);
    txtOut = (EditText)findViewById(R.id.txtOut);
    sb = (SeekBar)findViewById(R.id.seekBar);
    btn = (Button)findViewById(R.id.button);
```

You can rely on Android Studio's code assist to autocomplete most of these entries—all you have to do is type the first few characters, then click the entry from the list or press ENTER to accept the code recommendation. It's faster and will help you avoid common typing errors.

Connecting the Encode Button to the encode() Method

Now that we have the code connected to the components in the GUI layout, we can copy the `encode()` method from the desktop version of the app from Chapter 7. We'll paste the method into the `public class MainActivity` declaration, right below the five lines you added declaring the variables that point to the GUI components in the layout.

Open your *SecretMessagesGUI* desktop project in Eclipse and highlight the entire `encode()` method. Copy this code and paste it into the `public class MainActivity` declaration in Android Studio:

```
public class MainActivity extends AppCompatActivity {
    EditText txtIn;
    EditText txtKey;
    EditText txtOut;
    SeekBar sb;
    Button btn;
    public String encode( String message, int keyVal ) {
        String output = "";
```

```
        char key = (char) keyVal;
        for ( int x = 0; x < message.length(); x++ ) {
            char input = message.charAt(x);
            if (input >= 'A' && input <= 'Z')
            {
                input += key;
                if (input > 'Z')
                    input -= 26;
                if (input < 'A')
                    input += 26;
            }
            else if (input >= 'a' && input <= 'z')
            {
                input += key;
                if (input > 'z')
                    input -= 26;
                if (input < 'a')
                    input += 26;
            }
            else if (input >= '0' && input <= '9')
            {
                input += (keyVal % 10);
                if (input > '9')
                    input -= 10;
                if (input < '0')
                    input += 10;
            }
            output += input;
        }
        return output;
    }
```

Now that we have the `encode()` method in place, we just need to call that method whenever the Encode/Decode button is clicked. We'll create an `OnClickListener` for the Encode/Decode button and call the `encode()` method from there.

Inside the `onCreate()` method in *MainActivity.java*, start typing the code **`btn.setOnCl`** until a code recommendation list pops up, as shown in Figure 8-9. On the list, choose **`setOnClickListener()`**.

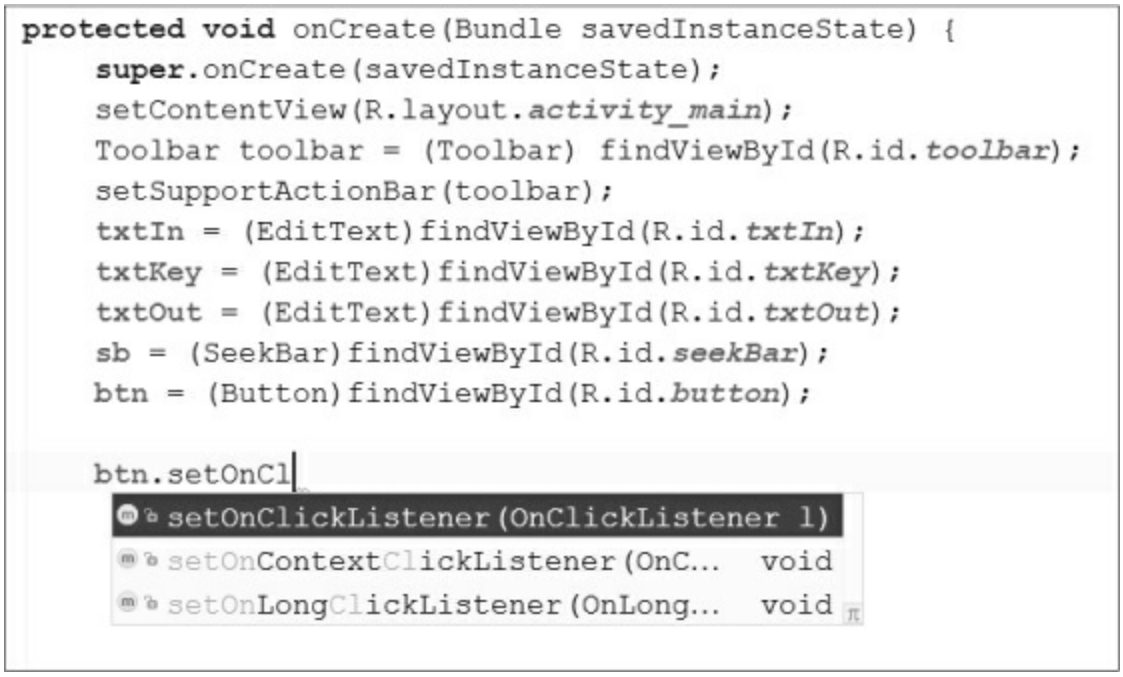


Figure 8-9: You can rely on code recommendations from Android Studio to autofill code for you, as shown here with the setOnClickListener() for the Encode/Decode button.

After clicking the `setOnClickListener()` code recommendation, click inside the parentheses of that method and begin typing **`new OnClickListener`**, as shown in Figure 8-10, double-clicking the first code recommendation that pops up:

btn.setOnClickListener(new OnC);
OnClickListener{...} (android.view.Vie
Press Ctrl+Shift+Space to show only variants that are suitable by type

Figure 8-10: Use the code assist once more to fill in the OnClickListener. This time, Android Studio provides several lines of code.

You'll notice that Android Studio fills in several lines of code for you for the `OnClickListener()` event handler:

```
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

These few lines create an event listener using an anonymous inner class of type `View.OnClickListener()`, with an `onClick()` event handler to respond when the user clicks the button.

Next we'll set up the `encode()` method. `encode()` doesn't care what's happening in the rest of our code as long as we give it two arguments that are the right data types. This means we don't have to change `encode()` to fit our new app, but we do need to set up new values to give it as arguments.

To encode the message the user typed into `txtIn`, we'll first need to retrieve the key value they entered into `txtKey`. Then, we'll get the string of text the user entered into `txtIn`. We'll pass both of those values to the `encode()` method as arguments and store the encoded result in a variable called `output`. Finally, we'll set the text of the `txtOut` field to the encoded output string `output`.

Add the following four lines within the braces of the `onClick()` method:

```
btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      ❶ int key = Integer.parseInt(txtKey.getText().toString());
      ❷ String message = txtIn.getText().toString();
      ❸ String output = encode(message, key);
      ❹ txtOut.setText(output);
      }
});
```

At ❶, we've used the `Integer.parseInt()` method to convert text from the user into an integer value `key`. The only difference between this `parseInt()` and others we've used before is that `txtKey` is followed by two methods: `getText()` and `toString()`. In Android, the `getText()` method on an `EditText` (that is, a text field) doesn't return a string directly—it returns a flexible type called `Editable`. All of the `EditText` types, from Plain to Password to Number, return an `Editable` object, because the text inside an `EditText` is meant to change and be editable by the user. We use the `toString()` method to turn the text from the `EditText` into a string that we can parse, to find the number the user entered.

The same is true at ❷: we need both methods, `getText()` and `toString()`, to get the input message from `txtIn` and turn it into a string that we can encode. At ❸, we use the `encode()` method on the `message` and `key` value the user provided to encode the message, then store the result in `output`. Finally, at ❹, we set the text of the `txtOut` text field to display the encoded `output` message.

Now, our app is functional enough for a first test. Let's run the app in the Android Emulator to see the Encode/Decode button at work.

Testing the App

Save your progress so far. Then, go to **Run ▸ Run 'app'**. Select an emulator you set up back in Chapter 4, such as My Nexus 6P, as shown in Figure 8-11.

Figure 8-11: Choosing an emulator

The emulator may take a few minutes to run the first time—remember, you can leave it running while working in Android Studio to avoid the long startup each time you want to run the app. You can also run the app directly on your Android device, as we did in Chapter 4, and skip the emulator if you wish.

When the emulator starts up and the Secret Messages app loads, you'll see the initial layout, as shown in Figure 8-12 (left). Type in a key value, then press the Encode/Decode button, and you'll see Android convert the message into ciphertext, as shown in Figure 8-12 (right).

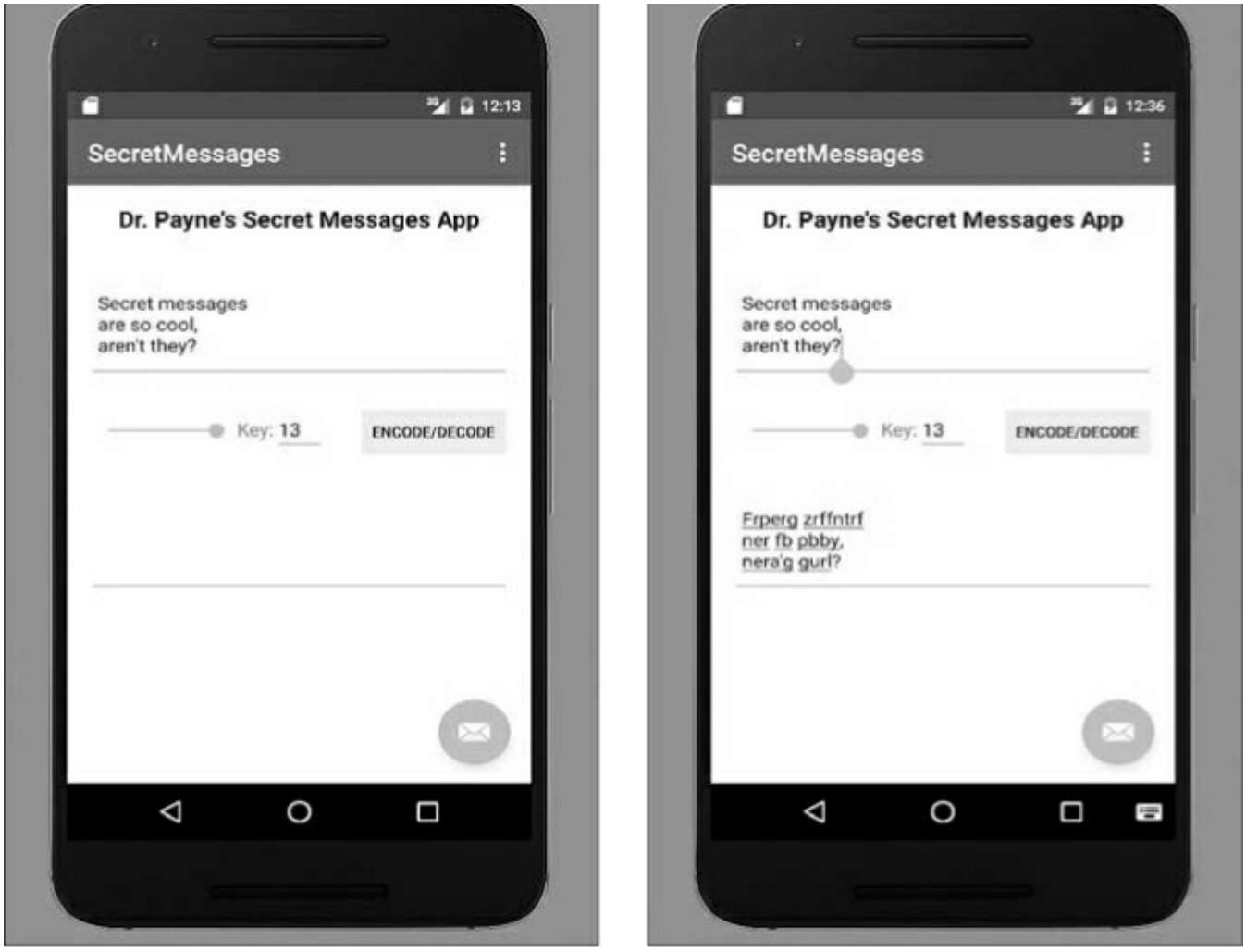


Figure 8-12: The Secret Messages app running on the Nexus 6P Android Emulator (left); clicking the Encode/Decode button creates the encoded text (right).

You can copy and paste between the fields, but the slider/`SeekBar` doesn't work yet. The floating action button—the email envelope symbol in the lower right—also doesn't do anything yet. Don't worry; we'll work on those shortly.

Notice there's one other user experience issue: the encoded message (`Frperg zrffntrf...`) is underlined because it looks like misspelled text to the computer. To turn off spellcheck, go back to the *content_main.xml* layout file and select the **txtOut** text field. Under the Properties pane, expand `inputType` and select the **textNoSuggestions** property. You can also turn off spelling suggestions for the `txtIn` text field, if you'd like, either separately or by selecting both text fields while holding down SHIFT in the layout view and then selecting the **textNoSuggestions** property. Now rerun the app on the emulator. No more spellcheck!

With this small tweak, we're ready to bring the `SeekBar` online so we can slide

among different key values easily and quickly.

Working with the SeekBar

We're going to wire the `SeekBar` to change the output message whenever the user slides left or right, so we need to modify a few of its properties to get it ready for use.

First, let's learn a little about the `SeekBar` widget. You've probably seen a `SeekBar` when playing a video on your mobile device. For example, on a YouTube video the `SeekBar` shows where you are in a video, and it allows you to skip forward or backward—this is sometimes called *seeking*.

The `SeekBar` differs from a `JSlider` in a few ways. First, a `SeekBar` can have only positive values. Zero is the minimum, but we can set the maximum to any positive value, like `26`. Also, a `SeekBar` doesn't show increment marks or labels on the bar itself. This means that if we want to allow our users to use negative key values, we'll have to use a bit of math to convert positive values to negative so that the user can slide the key value through the range of values from, say, `-13` to `+13`. To do this, we'll make the `SeekBar` range from `0` to `26`. And we'll need to update the `txtKey` field to show the values changing as the user slides the `SeekBar` to let them know the `SeekBar`'s current value. Let's begin by changing some of the properties of the `SeekBar` in the GUI layout.

Switch back to the *content_main.xml* layout file and select the `SeekBar` in the design preview. Under the Properties pane for the `SeekBar`, find `max` and `progress` and change both to **`26`**.

The `max` property is the maximum value the `SeekBar` will display. For the Secret Messages app, we need at least 26 values to cover the whole alphabet. We want the range to be `-13` to `+13` so that encoding and decoding is easy for the user. That would be 27 values total, so we'll choose `26` as the `max` (since that makes for 27 values when 0 is included). The `progress` property is the current value of the `SeekBar`, similar to the value property of a `JSlider`. We'll use the `getProgress()` method on the `SeekBar` to get the value the `SeekBar` is currently pointing to.

Save your changes and then switch back to the *MainActivity.java* source code file. Inside the `onCreate()` method, right below the closing line `});` of the `btn.setOnClickListener()` code, begin typing **`sb.set`** and use the code recommender to find `setOnSeekBarChangeListener()`. This will store your `SeekBar` in the variable `sb`.

The code for the SeekBar listener is different from that for the Encode/Decode button. We want to listen for more than just click events on the SeekBar. We want to listen for all changes, including the user pressing the SeekBar and sliding it left or right. For that, we need an OnSeekBarChangeListener.

Just as with the button code, though, we'll use the code recommender to automatically generate the new OnSeekBarChangeListener. Inside the parentheses after sb.setOnSeekBarChangeListener(), begin typing **new OnSeekBar** and use the code recommendation.

Android Studio will fill in the OnSeekBarChangeListener code with three methods: onProgressChanged(), onStartTrackingTouch(), and onStopTrackingTouch(). We want to use onProgressChanged(), as this will tell us when the user has changed the value of the SeekBar.

The steps we want to program into the SeekBar event listener are similar to the code for the Encode/Decode button, except that we need to calculate the key based on the SeekBar's value. We also need to show the key in the txtKey field. Add the following lines of code to the onProgressChanged() method:

```
sb.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean
fromUser) {
      ❶ int key = sb.getProgress() - 13;
        String message = txtIn.getText().toString();
        String output = encode(message, key);
        txtOut.setText(output);
      ❷ txtKey.setText("" + key);
}
```

At ❶, we create a variable named key. We get the progress value from the SeekBar sb variable, and we subtract 13 from that value. The SeekBar can go only from 0 to 26, so subtracting 13 will give a range from -13 to +13, perfect for encoding and decoding messages. The next three lines are identical to those in the Encode/Decode button event handler code, which get the message, encode the message using the encode() method, and then display the encoded text in txtOut.

The line at ❷ is new. When the user moves the SeekBar slider, we want the txtKey readout to change. Since key is an integer, we add an empty string ("") to it to convert the result to a text string that will be displayed to the user. You could have also used the command Integer.toString(key) to convert the key value

into a string using the `Integer` class's `toString()` method, but adding an empty string to an `int` is just a handy shortcut.

Running the App on the Emulator and on an Android Device

Save your changes and then try running the app again on the emulator. You can now slide the `SeekBar` left and right and see the ciphertext change immediately with each key value, as shown in Figure 8-13.

It's been a few chapters since we connected a physical Android device, so let's review that quickly. First, connect a compatible USB cable between your Android device and your computer. When a pop-up appears asking whether you want to allow USB Debugging, click **Yes**.

Figure 8-13: The SeekBar is now fully functional, as shown on the Android Emulator, with a key value of 5.

In Android Studio, click the run button, and you'll see a Select Deployment Target dialog. (You may need to close the app in your emulator window to get the dialog to show up.) Find the entry for the Android device you've connected and click **OK**.

When you run the app on your real Android device, you'll notice the `SeekBar` is more responsive, and you have access to a few options when you click certain parts of the app. When you click the ciphertext and choose **Select All**, a context menu will appear, as shown in Figure 8-14, offering you the option to Cut, Copy, Share, or Assist.

If you click Share, the selected text can be sent via email, text message, chat, Bluetooth, Twitter, Facebook, or any other app you have configured for sharing on your device, as shown in Figure 8-15 (left). With the Share option, you can

send encoded messages to anyone from inside the app! You can even email a secret message to a friend who has the desktop GUI version of the Secret Messages app, as shown in Figure 8-15 (right). All they have to do is copy the ciphertext from the email you send them into the Secret Messages GUI app while it's running, then use the opposite of the secret key to decode the message: for example, `-8` would decode a message sent with a key value of `8`.

Figure 8-14: The context menu allows you to Cut, Copy, or Share your encoded messages.

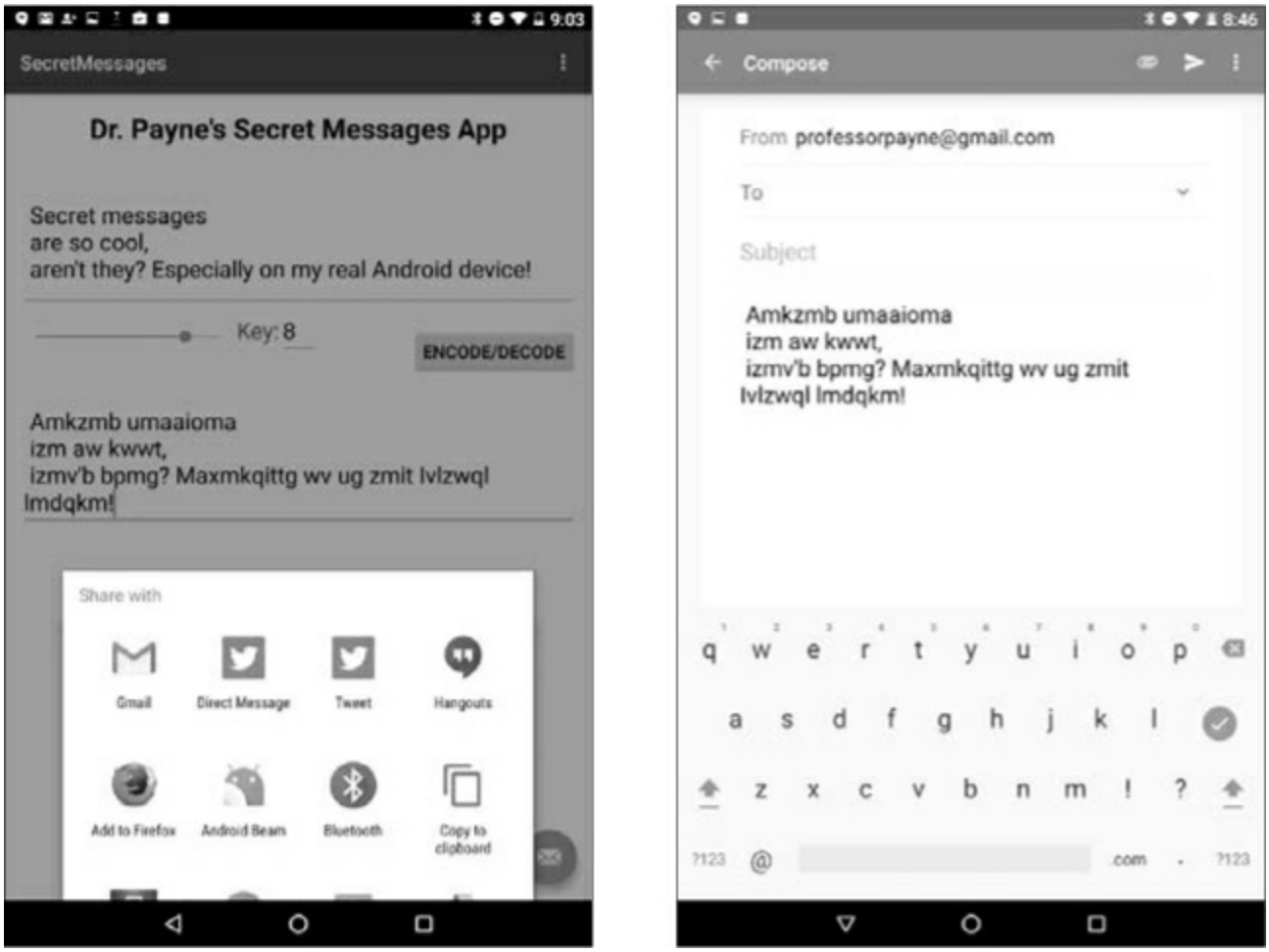


Figure 8-15: The Share option allows you to share your message via email, text message, Twitter, Facebook, or any other app you've configured for sharing (left). Selecting the email option will automatically create an email message with your encoded text (right).

So now we have a fully functional Secret Messages mobile app that allows us to share our messages by email, text, or social media with anyone.

There's only one feature left to make our app complete: adding our own custom action to the floating action button. This will enable us to let the user share their messages at the touch of a finger!

Bonus: Customizing the Floating Action Button

So far, we've been ignoring the floating action button (also known as the *fab icon*), the circular email envelope symbol in the lower-right corner of the app screen, shown in Figure 8-16.

The fab icon gives us a way to provide the user with a shortcut to perform a specific action, such as sharing their message via email, text, or social media.

Figure 8-16: The floating action button

Android Studio automatically includes some code for the fab icon that displays a new type of pop-up message—called a "Snackbar" in Android lingo—in the `onCreate()` method:

```
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Replace with your own action",
Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
    }
});
```

All we need to do is replace the code in the `onClick()` method of the `fab.setOnClickListener()` with code that does what we want.

First, delete the code between the braces of the `onClick()` method. Next, type the following code into the `onClick()` method:

```
FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
     ❶ Intent shareIntent = new Intent(Intent.ACTION_SEND);
     ❷ shareIntent.setType("text/plain");
     ❸ shareIntent.putExtra(Intent.EXTRA_SUBJECT, "Secret Message "+
              DateFormat.getDateTimeInstance().format(new Date()));
     ❹ shareIntent.putExtra(Intent.EXTRA_TEXT, txtOut.getText().toString());
     ❺ try {
          startActivity(Intent.createChooser(shareIntent, "Share
message..."));
          finish();
       }
     ❻ catch (android.content.ActivityNotFoundException ex) {
          Toast.makeText(MainActivity.this, "Error: Couldn't share.",
                  Toast.LENGTH_SHORT).show();
       }
    }
});
```

Now let's look at what each line does. At ❶, we create an `Intent` called `shareIntent`. An `Intent` in Android is an activity that we want to start or launch, such as email, Twitter, or the camera. The `shareIntent` uses `ACTION_SEND`, which means we want to send some information to another app on the Android device.

We set the type of the `Intent` to `text/plain` ❷, which is what we need for an email, tweet, or social media post, and then add a subject line to the activity ❸. If the user chooses to share their message by email or via another app that uses a subject line, it will read "Secret Message" plus the date. The `DateFormat` class will format the date in text form based on your region, like `"Dec 7, 2017 4:33:12 PM"`.

The line at ❹ sends the encoded message from `txtOut` as the body of an email, tweet, or post, depending on which app the user chooses to share their message through.

Next is a `try` statement ❺. Lots of things could go wrong when we try to call one app from inside another. What if the user doesn't have an email app installed? What if the app is busy or the network can't be reached? The `try` statement is made for situations like this. The first line of code inside the `try` statement attempts to start the activity the user selected (like email, text messaging, or Twitter), then passes the information to the selected app, and finally finishes the activity in the next line.

If there's an error or exception, the `catch` statement ❻ will display a Toast message (a kind of pop-up) that reads `"Error: Couldn't share."`.

That's all we need to get the fab icon working. Remember to import new classes by pressing ALT-ENTER (or OPTION-ENTER) while typing the lines of code. Remember, if Android Studio underlines a new class in red, it's often because the class hasn't been imported, so pressing ALT-ENTER (or OPTION-ENTER) will automatically add the correct `import` statement, just as we did in Chapter 4.

Once you've added the code and imported the classes, run the app again.

After you encode a message, hide the keyboard by pressing the down triangle at the bottom of your screen. Then, tap the fab icon on the bottom right of the screen, as shown in Figure 8-17 (left). Select your email app from the apps under **Share message...**, and the app should start, with a subject line and the encoded message in the body of the email, as shown in Figure 8-17 (right).

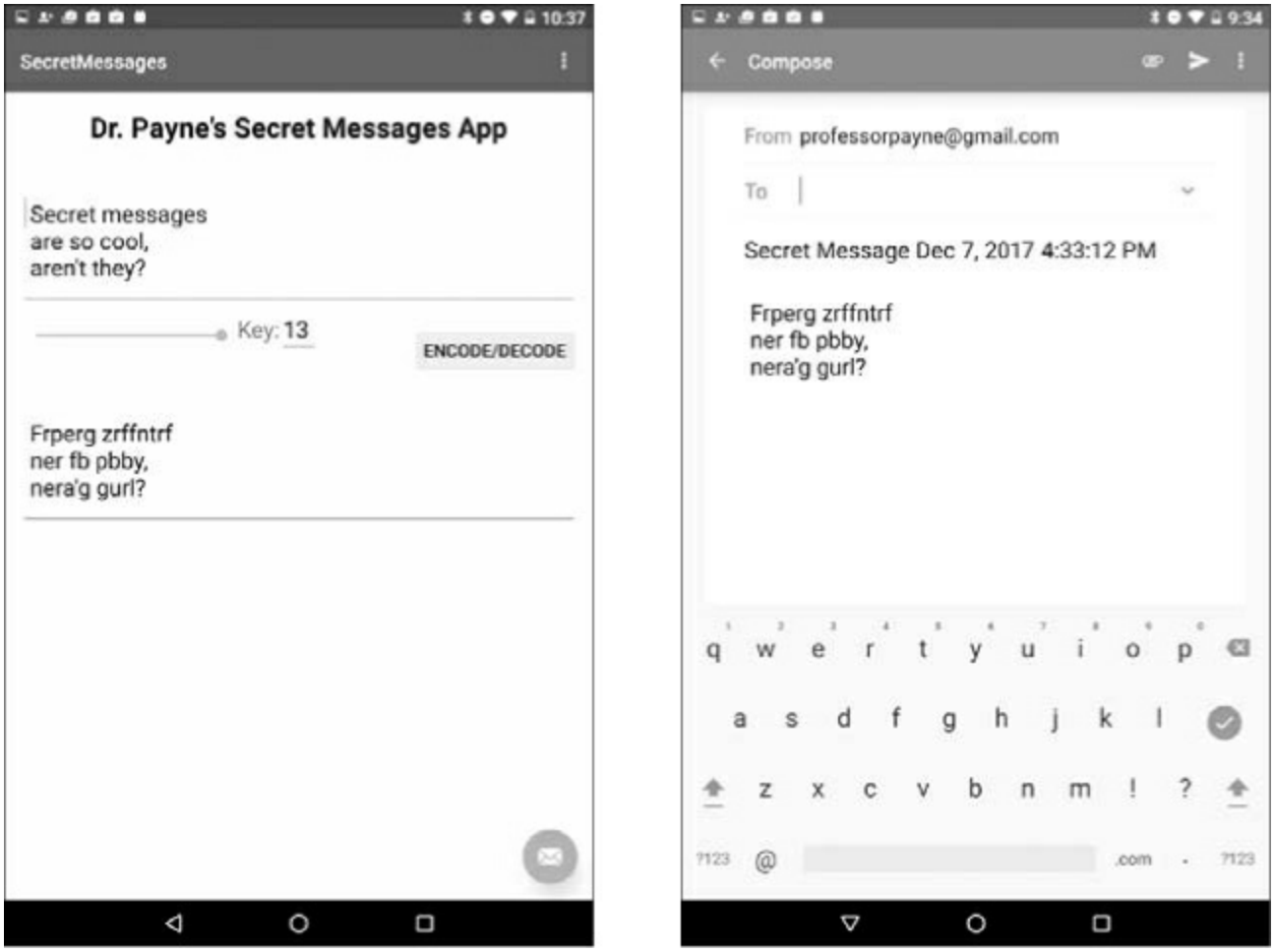


Figure 8-17: Pressing the fab icon in the lower right pops up a sharing menu (left). Choosing the email app from the sharing menu automatically creates an email with the secret message as the body and today's date and time in the subject line (right).

Now we can send secret messages from the app with just the touch of an icon! But to make this app truly worthy of the Google Play Store, we need to give it the ability to *receive* secret messages from outside the app.

Receiving Secret Messages from Other Apps

Let's make one last upgrade to the mobile version of the Secret Messages app. We can share, email, or tweet encoded messages from the app, but what if we want to *receive* text *from* emails, tweets, and posts without copying and pasting into the app? To address this, we can add our app to the Share list in Android so that other apps can send text directly to the Secret Messages app—and it only takes a few lines of code!

This is an advanced technique, so let's walk through the process step-by-step.

First, open your app's *AndroidManifest.xml* file under **manifests ▸ AndroidManifest.xml** in the Project Explorer pane. The manifest file stores the "shipping" information for your app, much as a ship's manifest lists all the cargo traveling on the ship. You can declare properties for your entire app and include the details an Android device needs to run the app, just as you can declare the contents of a crate or package when you ship it overseas.

In order to do this, you'll need to edit some XML. XML files contain *tags* to label information held in the file. Usually a tag is formatted with its name between angled brackets, like `<tag>`. In this case, we need to modify the manifest's `intent-filter` tag entry, which has information about what data the app will accept. Add the following five lines of XML near the end of your *AndroidManifest.xml* file, before the `</activity>` tag:

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        ❶ <intent-filter>
            ❷ <action android:name="android.intent.action.SEND" />
            ❸ <category android:name="android.intent.category.DEFAULT" />
            ❹ <data android:mimeType="text/*" />
        ❺ </intent-filter>
    </activity>
</application>
</manifest>
```

At ❶, we add a new `intent-filter` to the app. Earlier we used an `Intent` object to send text out to another app, such as email, Twitter, or Facebook. In this case, we're creating a filter to *receive* `Intent` objects sent from other apps on our phone ❷. (We used `Intent.ACTION_SEND` in the fab icon code on page 191 to send `Intent` messages to other apps; here we're doing the opposite.)

The line at ❸ uses the `DEFAULT` sharing category to add the Secret Messages app to the list of apps on your phone that you can share with. With the `DEFAULT` category, your app can receive data from any app. (There are other categories for when you want to filter or restrict the types of apps that can send data to your app. For example, the `BROWSABLE` category is used by apps that should receive data from only a web browser.) At ❹, we tell Android that the Secret Messages app should receive any kind of text data from other apps. (If you write an app that accepts images as input, your `mimeType` might be `"image/*"`, or `"image/png"` for

just PNG images. For video, there's `"video/mp4"`, and so on.)

Finally, we close the `intent-filter` entry ❺ in your app's manifest. XML tags need to be *closed*, which is usually done with a forward slash and the name of the XML tag that's being closed, as in `</intent-filter>`. This lets the computer know you've completed code for the `intent-filter`. Save your *AndroidManifest.xml* file and then open the *MainActivity.java* file to begin editing the Java source code.

Scroll down inside *MainActivity.java* to find the `onCreate()` method. Immediately after the code that connects the GUI components to the variables from `txtIn` through `btn`, add the four statements shown here:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    txtIn = (EditText)findViewById(R.id.txtIn);
    txtKey = (EditText)findViewById(R.id.txtKey);
    txtOut = (EditText)findViewById(R.id.txtOut);
    sb = (SeekBar)findViewById(R.id.seekBar);
    btn = (Button)findViewById(R.id.button);
  ❶ Intent receivedIntent = getIntent();
  ❷ String receivedText = receivedIntent.getStringExtra(Intent.EXTRA_TEXT);
  ❸ if (receivedText != null)
      ❹ txtIn.setText(receivedText);
    btn.setOnClickListener(new View.OnClickListener() {
```

At ❶, we create a variable called `receivedIntent` to accept the incoming `Intent` message shared from another app. Then, we get the text from the `Intent` message by calling the `getStringExtra()` method ❷. The parts of a message sent by an `Intent` are referred to as *extra* data. An `if` statement ❸ checks to make sure the `receivedText` variable isn't `null` and, if so, changes the text in the `txtIn` text field at the top of the Secret Messages app to the received text ❹.

Believe it or not, that's all the work necessary to accept messages shared from other apps into the Secret Messages app! Now, you can not only send encoded messages from the app but also decode messages from email, tweets, and posts directly. Figure 8-18 shows what it looks like to share a message from email to the Secret Messages app for quick decoding.

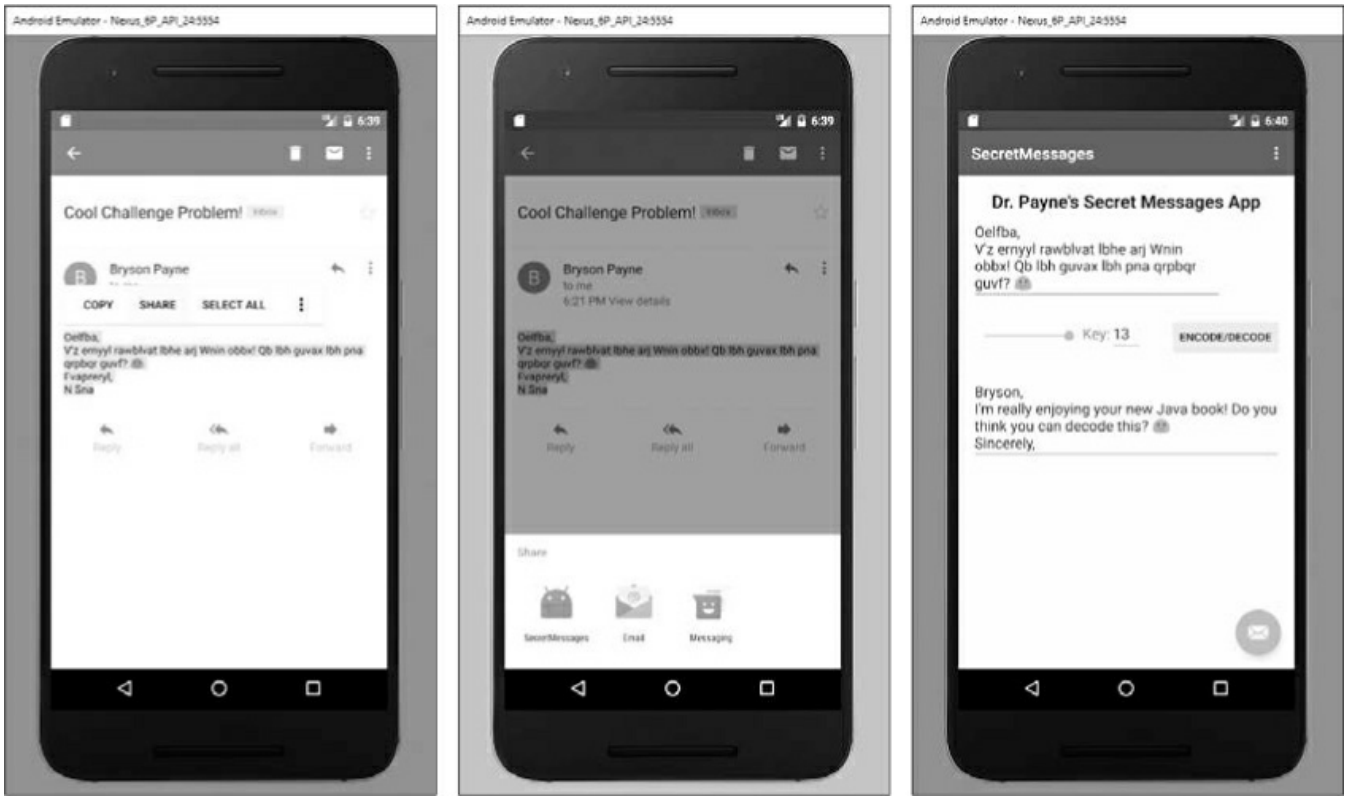


Figure 8-18: Now the Secret Messages app can send and receive messages back and forth with other apps on your phone, including your email.

Select text in your favorite email app, or in Twitter or Facebook, and press the **Share** button. The Secret Messages app will appear on the list of apps that accept a shared message. Select **SecretMessages** as the app to share with, and the selected text will be shown in the input text field of the Secret Messages app. You can encode or decode with ease; just remember you may need to change the key value depending on how the message was encoded.

Congratulations! You've built a fully social Secret Messages mobile app. Enjoy it and share it with a friend!

Let's take a look at the skills you developed in this chapter and try a couple of additional programming challenges.

What You Learned

We reused code in this app from the GUI desktop version of the Secret Messages app, and we added so much more to make this a truly complete mobile app. Here are some of the new things you've learned and some of the skills you've reinforced in this chapter:

- Writing event handlers for Android components, including buttons, `SeekBars`, and the floating action button
- Customizing Android GUI components
- Using multiline text inputs in an Android app
- Adding a `SeekBar` and using it like a slider interface
- Running Android apps on both the emulator and real devices
- Customizing the action of the floating action button
- Creating custom `Intent` objects to call special activities, including sending data to other apps, from inside an app
- Creating custom `Intent` filters to receive text (or image or video) data sent from other apps

Programming Challenges

Try these programming challenges to review and practice what you've learned, and to expand your programming skills by trying something new. Visit the book's website at *https://www.nostarch.com/learnjava/* for sample solutions.

#1: Creating a Move Up Button

This programming challenge is to create a Move Up ^ button similar to the one in Chapter 7 (Programming Challenge #1 on page 172) that moves text up from the output field into the input field.

Place the Move Up ^ button below the `txtOut` text field. Then add an event handler that gets the text from the `txtOut` text field and sets it as the text of the `txtIn` text field. As an added bonus, you could have the event handler for the Move Up ^ button also change the `SeekBar` to the opposite of its value. An encryption key of `7` would become `-7`, the decryption key. The result will be that clicking the button swaps the encoded message up and automatically decodes it!

#2: Changing the SeekBar's Progress

A second improvement you could make to the user experience would be to change the `SeekBar`'s progress property whenever the user enters a key value

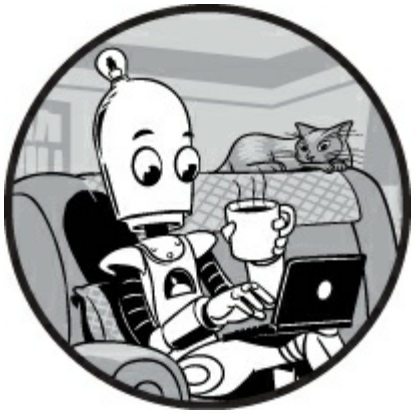
manually in the `txtKey` field. Give it a try!

HINT

Instead of creating an event handler for the txtKey field, modify the code for the Encode/Decode button to change the SeekBar's progress to match the value in txtKey + 13—remember the SeekBar goes from 0 to 26, not -13 to +13. One or two lines of code should be all you need.

9

PAINT COLORFUL BUBBLES WITH YOUR MOUSE!



In the next three chapters, we'll build BubbleDraw, an interactive, animated drawing app that lets the user draw floating, bouncy, colorful bubbles with their mouse in the desktop version and with their fingers in the mobile version!

The first version of the BubbleDraw app will look like Figure 9-1, with each bubble in a different random color. It will let the user draw bubbles of random colors whenever they click and drag the mouse on the app window. The user will also be able to change the size of the bubbles by scrolling the mouse wheel up and down or by using a scroll gesture on a trackpad or touchscreen.

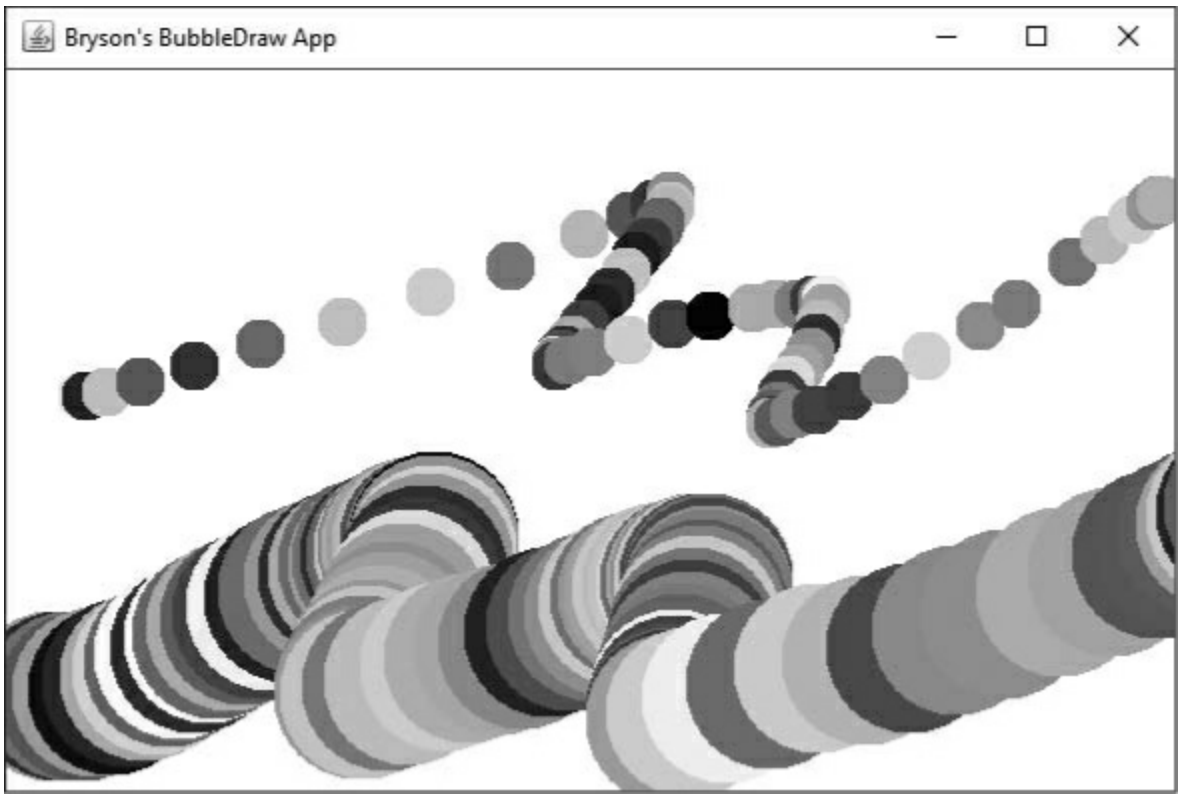


Figure 9-1: The BubbleDraw app lets the user draw randomly colored bubbles using the mouse.

We're going to use an object-oriented programming approach to build the BubbleDraw app. The variables, functions, loops, and conditional statements you've learned so far are known as *procedural programming*. Procedural programming involves writing programs step-by-step in a linear fashion, sort of like following a recipe. Object-oriented programming uses all of these concepts, but it allows us to program much larger and more complicated applications by breaking a big software project into smaller pieces called *objects*.

As an example, a bubble is an important entity in the BubbleDraw app, so bubbles might become objects as we begin to code this app. First, we'll tackle the problem of defining what a bubble is. Second, we'll figure out a way to store information on lots of bubbles and how to draw the bubbles on the screen. Finally, we'll add the ability to click and drag the mouse to create bubbles. Instead of writing separate pieces of code to control each bubble individually, we'll write one piece of code that applies to all bubble objects.

We'll begin with two source code files: one for the app window called *BubbleDraw* and one for the drawing canvas called *BubblePanel*. The app window will extend a familiar class, `JFrame`, and the drawing canvas inside the window frame will use a new kind of GUI container called a `JPanel`. By building two

separate files, we'll be able to reuse the drawing canvas in a GUI app in Chapter 10. Let's start coding!

Creating the BubbleDraw Project Files

In Eclipse, go to **File ▸ New ▸ Java Project** and create a new project folder for the BubbleDraw app. Name the project **`BubbleDraw`** and click **Finish**.

Expand the *BubbleDraw* project folder in the Project Explorer pane, right-click the *src* folder, and select **New ▸ Class**. Create a class called `BubbleDraw` for the app window, with `javax.swing.JFrame` as the superclass, and under **Which method stubs . . .** select the **public static void main(String[] args)** checkbox. Then click **Finish**.

Next, we'll create the BubblePanel drawing canvas. Right-click the *src* folder and select **New ▸ Class**. Name this class **`BubblePanel`**, with a superclass of `javax.swing.JPanel`. Click **Finish**.

Using these classes, we'll create a reusable `BubblePanel` canvas that we can expand and put into other apps. The `BubbleDraw` frame will be a container that displays the `BubblePanel`.

Building the BubbleDraw Frame

Let's begin by setting up the main app window in the *BubbleDraw.java* source code file. Click the *BubbleDraw.java* tab at the top of the content pane in Eclipse, and you'll see the following code has already been filled in:

```
import javax.swing.JFrame;
public class BubbleDraw extends JFrame {
    public static void main(String[] args) {
    }
}
```

The app window runs in a `JFrame`, like our previous GUI apps did. In this app, we need the window to display the drawing canvas, our `BubblePanel`. We won't add any other GUI components to this first version of the app.

We want to create a `JFrame` window and add setup code for the frame as we have in previous chapters, but we'll also add a `BubblePanel` to the frame for our

drawing canvas. The complete `BubbleDraw` class will look like Listing 9-1:

```
import javax.swing.JFrame;
public class BubbleDraw extends JFrame {
    public static void main(String[] args) {
      ❶ JFrame frame = new JFrame("Your Name's BubbleDraw App");
      ❷ frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      ❸ frame.getContentPane().add(new BubblePanel());
      ❹ frame.setSize(new java.awt.Dimension(600,400));
      ❺ frame.setVisible(true);
    }
}
```

Listing 9-1: The BubbleDraw class

Start by creating a new `JFrame` with a title bar at the top ❶. Put your own name in the title string. Next, set the default close operation so that the app exits when the user closes the window ❷. Then add the code to open a new `BubblePanel` drawing canvas as the content inside this frame ❸. The last two lines set the size of the window ❹ and make it visible ❺.

Save the *BubbleDraw.java* file and run it once. You'll see a gray Java frame with "*Your Name*`'s BubbleDraw App`" in the title bar. We'll build the logic for the `BubblePanel` drawing canvas next.

Creating a Class for Bubbles

Switch to the tab for the *BubblePanel.java* file. This file will contain all the logic for drawing bubbles on the screen. Our first task will be creating a `Bubble` class to store each bubble's color, size, and location on the screen.

Defining a Bubble

The reason for building a class is quite practical: in the older, procedural approach to programming, we would need a separate variable for each bubble's x- and y-coordinates, size, and so on. For example, we'd need a `bubble1x` variable for the first bubble's x-location, as well as `bubble1y`, `bubble1size`, and `bubble1color` variables for that bubble. That wouldn't be so bad, but what if we drag the mouse for a few seconds and wind up with a thousand bubbles? We would need

4,000 variables just to keep track! Instead, we can use properties inside a class to store those values for each bubble.

Before we start writing code, let's define what a bubble is. As shown earlier in Figure 9-1, a bubble is a colorful, filled circle, and bubbles can be different sizes and in different locations.

All of those features are attributes of bubbles in this app. In object-oriented programming, we use the nouns and adjectives that describe an object to build a list of attributes when we create a new class. Attributes are stored as class variables.

Besides attributes, a class can also contain methods. A method is a type of function associated with a specific class. Methods enable a class to do something. Think about what we want the bubbles in the drawing app to do. We need to create a bubble whenever the user clicks or drags the mouse. We'll also draw a bubble every time the screen refreshes. The verbs—like *create* and *draw*—will help us identify the methods we need in our bubble class.

When we capture all of those attributes and methods into a single class, we can describe any bubble we might want to show on the screen. This is how a coder solves problems with object-oriented programming: by breaking big applications into smaller pieces, creating classes by asking questions about what the program contains, and then identifying methods and attributes by asking what each object of a new class should do and what kind of information it needs.

Let's begin writing the `Bubble` class. We'll do this inside the `BubblePanel` class, as `BubblePanel` is the only place we need to draw bubbles. Start coding your `Bubble` class before the closing brace at the end of `BubblePanel`, as follows:

```
import javax.swing.JPanel;
public class BubblePanel extends JPanel {
    private class Bubble {
    }
}
```

We usually make internal and helper classes private to keep other programs from accessing them directly. So, to make `Bubble` an internal class accessible only from a `BubblePanel`, we declare it as `private`. This technique, called *encapsulation*, means that the `BubblePanel` class hides its inner workings from other classes. Encapsulation is one of the core principles of object-oriented programming, and it's a best practice worth following because it means we can change the way the `Bubble` class works internally and know for certain that we won't break other

parts of the code. Since `Bubble` is declared as `private`, we know that there isn't any code outside of the `BubblePanel` class that depends on it. This is especially important for bigger applications that contain a large number of classes and are worked on by many programmers.

`BubblePanel` will rely on the `Bubble` class to store information on individual bubbles as the user draws on the screen. A bubble has an (x, y) location on the screen, a size, and a color, so we can make these attributes as variables inside `Bubble`. The x- and y-coordinates of the bubble's location and the size of the bubble can all be stored as integer values:

```
import javax.swing.JPanel;
public class BubblePanel extends JPanel {
    private class Bubble {
        private int x;
        private int y;
        private int size;
    }
}
```

Here we make two separate variables called `x` and `y` to store the bubble's coordinates, as well as a `size` variable. These attributes are `private`, so only the `Bubble` class itself can change those values directly. We encapsulate all the data a bubble should have inside the `Bubble` class, and we interact with bubbles using the methods in that class only.

As mentioned earlier, each bubble in our app can have its own color. The `java.awt` library has a `Color` class that handles *RGB* (*red-green-blue*) color values to reproduce any color that we can display on our monitor. We'll talk a bit more about RGB when we actually write the code to control bubble colors later in the chapter, so for now just import the `java.awt.Color` class at the top of the file and add an attribute called `color` to the `Bubble` class, like so:

```
import java.awt.Color;
import javax.swing.JPanel;
public class BubblePanel extends JPanel {
    private class Bubble {
        private int x;
        private int y;
        private int size;
        private Color color;
    }
}
```

Now that we've added the four attributes we want each bubble to have (`x`, `y`,

size, and color), we can start building the behaviors, or functions, of a bubble to give values to these attributes. We'll do that by adding methods inside the Bubble class.

Designing a Bubble's Methods

The Bubble class will have two actions, which we'll turn into methods: creating a bubble and drawing a bubble on the desktop screen.

Methods that create objects have a special name: *constructors*. A constructor sets up an object by assigning values to the object's attributes. When we assign values to the attributes inside a constructor, we're *initializing* them. For the Bubble class, we want to initialize the x, y, size, and color attributes of an individual bubble.

Building the Constructor

A constructor method begins with the keyword public, followed by the name of the class and a pair of parentheses. If there are any parameters you want to pass in as you set up an object, you place those inside the parentheses. We want to assign each bubble its x, y, and size values when it is created, so the constructor for the Bubble class looks like Listing 9-2.

```
private class Bubble {
    private int x;
    private int y;
    private int size;
    private Color color;
  ❶ public Bubble(int newX, int newY, int newSize) {
     ❷ x = newX;
     ❸ y = newY;
     ❹ size = newSize;
    }
}
```

Listing 9-2: The Bubble class constructor

Each bubble's coordinates and size will be determined by the user, so we want to pass the x- and y-coordinates and a size as integer values to the Bubble() constructor as each new bubble is created. These values are the three parameters we pass to the Bubble() method and are called newX, newY, and newSize ❶. Later in

the code, when we handle the user creating bubbles, we'll be outside the private `Bubble` class. That means we won't have direct access to the bubble's attributes, so we can't just assign values to the `x`, `y`, and `size` attributes of each bubble. In order to get around that, we assign the attributes values through the `Bubble()` constructor, which takes the input from the user; assigns the input values to `newX`, `newY`, and `newSize`; and then assigns the user input values to the bubble's attributes at ❷, ❸, and ❹.

We also want each bubble to have a random color. Because the color will be random, we don't need to pass it to the constructor using a parameter. Instead, we can create the color inside the constructor.

We'll need to generate random RGB color values to create a color for each bubble. RGB works by combining different amounts of red, green, and blue light on your computer monitor to produce different colors. In programming, each of these three colors is represented by an integer between 0 (none of the color) and 255 (the maximum amount of the color possible). To make a color, we need to get three integers between 0 and 255, separate them with commas, and group them together between parentheses. For example, pure red has an RGB value of (255, 0, 0), meaning full red, no green, and no blue. Yellow is (255, 255, 0) in RGB, meaning full amounts of red and green, but no blue. In total, there are over 16 million RGB color values that can be created this way.

We've generated random numbers with the `Math` class's `Math.random()` method before when we needed a single number, but in the BubbleDraw app, we'll need three random numbers (one for each part of RGB), so let's look at a new way to generate random values.

The `java.util.Random` class contains several useful methods, including `nextInt()`, which will let us instantly generate a random integer up to a certain size without having to do additional math or rounding. To use the `Random` class, we'll have to import it first, at the top of the *BubblePanel.java* file:

```
import java.util.Random;
import java.awt.Color;
import javax.swing.JPanel;
```

Because `Random` is a class, we need to create a new object or variable of the `Random` type to access its functions. Add the following line inside the top of the `BubblePanel` class:

```
import java.util.Random;
```

```
import java.awt.Color;
import javax.swing.JPanel;
public class BubblePanel extends JPanel {
    Random rand = new Random();
    private class Bubble {
```

This line creates a random number generator called `rand` that will allow us to generate random integer or floating-point values quickly. We're encapsulating `rand` by putting it at the top of the `BubblePanel` class so that we can generate random numbers both in the drawing window and in the inner `Bubble` class, but not outside a `BubblePanel`.

To create a random color in Java, we can use the constructor for the `java.awt.Color` class that we imported earlier. The constructor will accept three integer arguments between `0` and `255` for the red, green, and blue values to create a color.

Add the following code to the `Bubble()` constructor method:

```
    public Bubble(int newX, int newY, int newSize) {
        x = newX;
        y = newY;
        size = newSize;
        color = new Color( rand.nextInt(256),
                rand.nextInt(256),
                rand.nextInt(256) );
    }
```

We use the `new` keyword with the constructor for the `Color` class to create a new color, and each of the RGB color values will be a randomly generated integer between `0` and `255`. Every time you call the `nextInt()` method, it generates a new random integer between 0 and the maximum integer you pass it. Here, we want random color values from `0` to `255`, so we pass it `256` because `nextInt()` generates an integer up to, but not including, the upper bound passed to it.

Coding the Method to Draw a Bubble

Now that each bubble has an (x, y) location, a size, and a random color, let's add the ability to draw bubbles on the screen. To draw colorful graphics on the screen, we'll import the `java.awt.Graphics` class. This `Graphics` class contains methods such as `setColor()` to choose a paint color, `drawRect()` to draw a rectangle, and `fillOval()` to paint a filled-in oval on the screen, just to name a few. We'll use `fillOval()` to draw our circles, so add the following `import`

statement to the top of the *BubblePanel.java* file:

```
import java.awt.Graphics;
import java.util.Random;
import java.awt.Color;
import javax.swing.JPanel;
```

Next, let's add a `draw()` method to the `Bubble` inner class, below the `Bubble()` constructor:

```
    private class Bubble {
        private int x;
        private int y;
        private int size;
        private Color color;
        public Bubble(int newX, int newY, int newSize) {
            --snip--
        }
        public void draw(Graphics canvas) {
            ❶ canvas.setColor(color);
            ❷ canvas.fillOval(x - size/2, y - size/2, size, size);
        }
}
```

The `draw()` method accepts one parameter, a `Graphics` canvas called `canvas`. Inside the `draw()` method, we call the `setColor()` method on `canvas` ❶ to set the drawing color to the color stored in the `color` variable of the bubble we'll draw. At ❷, we call `fillOval()` to paint a filled-in circle on the screen. The `fillOval()` method takes four parameters, the x- and y-coordinate of the upper-left corner of a bounding box in which we'll draw the oval, followed by the width and height of the box. Think of the bounding box as an invisible rectangle at that (x, y) location with the given width and height and the oval as a balloon that will be blown up until it touches all four sides of the box (see Figure 9-2). By setting the same width and height for the box to make it square, we'll end up with a circle instead of an oval.

Because our ovals are actually circles, the width and height are the same value —the size of the bubble in pixels we stored in the `size` attribute. At ❷, we want the bubble to be centered on the (x, y) location the user clicked, so we adjust the upper-left corner of each bubble by subtracting half the size (`size/2`) from each of the `x` and `y` values.

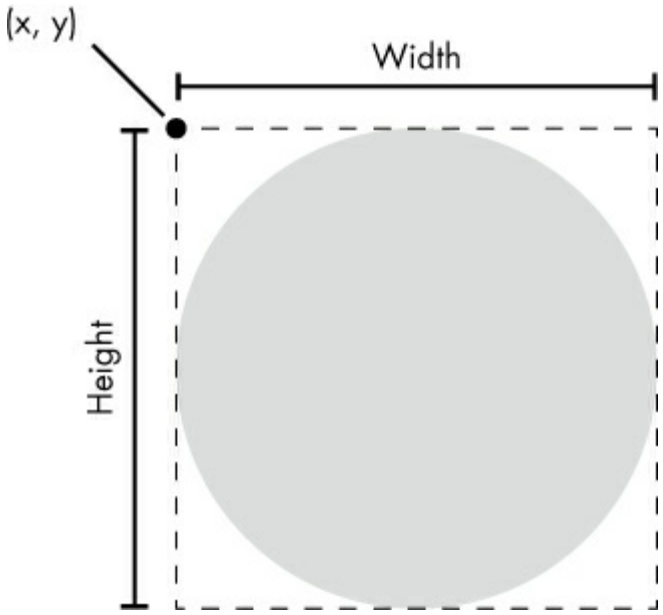


Figure 9-2: The fillOval() method takes four parameters, the upper-left corner's xand y-coordinates, followed by the oval's width and height.

With the addition of the `draw()` method, we now have a complete `Bubble` class that can remember the location, size, and color of a bubble. The class methods we made can be used to create new bubbles and draw them on the screen. It's time to add logic to the `BubblePanel` class to use those methods to create and draw bubbles whenever the user clicks and drags on the screen.

Storing Bubbles in an ArrayList

We need a way to store all the bubbles the user creates when they click and drag around the screen. Java's libraries contain several useful *data structures*, which are classes for storing groups of objects.

The `java.util.ArrayList` is a *dynamic* data structure. That means it not only stores a collection of objects but can also grow or shrink depending on the needs of the program. For the BubbleDraw app, we can't predict how many bubbles the user will draw, so a dynamic data structure like an `ArrayList` is the perfect option for storing all the bubbles a user could create. An `ArrayList` is a flexible way to store things when you don't know how many items you'll need in advance. A regular array in Java has a fixed size, but an `ArrayList` can keep adding new bubbles every time the user clicks.

First, let's import `java.util.ArrayList` at the top of the *BubblePanel.java* file so we can access `ArrayList` data types:

```
import java.util.ArrayList;
import java.awt.Graphics;
import java.util.Random;
import java.awt.Color;
import javax.swing.JPanel;
```

Next, we need to declare an `ArrayList` that will hold objects of the `Bubble` class. Inside the `BubblePanel` class, add the following declaration:

```
public class BubblePanel extends JPanel {
    Random rand = new Random();
    ArrayList<Bubble> bubbleList;
```

The `ArrayList` data structure can accept a type specifier inside the angle brackets, `<` and `>`, to tell Java what type of objects the `ArrayList` will hold. An `ArrayList` can hold any type of object, but the `ArrayList` we're declaring, `bubbleList`, will contain only objects of the `Bubble` class.

Next, let's take care of one additional variable, `size`, for the default bubble size:

```
public class BubblePanel extends JPanel {
    Random rand = new Random();
    ArrayList<Bubble> bubbleList;
    int size = 25;
```

This `size` variable sets a starting size for bubble diameters in pixels. I picked `25` pixels, but you can choose a larger or smaller starting size if you'd like.

Adding a Constructor to the BubblePanel Class

We've declared `bubbleList` to be a dynamic `ArrayList` of `Bubble` objects, so let's add a constructor to the `BubblePanel` class to initialize `bubbleList` and set the background color of the drawing window.

Just as we saw in the `Bubble` inner class, the constructor for the `BubblePanel` class uses the format `public` plus the name of the class, followed by a pair of parentheses:

```
public class BubblePanel extends JPanel {
    Random rand = new Random();
    ArrayList<Bubble> bubbleList;
    int size = 25;
    public BubblePanel() {
    }
```

Notice that we end the constructor with open and close parentheses, because the constructor is a method. Inside the constructor, we initialize `bubbleList` to get it ready to store a flexible `ArrayList` of `Bubble` objects. We also set the background color of the drawing window to black:

```
public BubblePanel() {
 ❶ bubbleList = new ArrayList<Bubble>();
 ❷ setBackground(Color.BLACK);
}
```

NOTE

Although I recommend a black background for the app, the images in this book show a white background for readability.

Just as in the declaration of `bubbleList`, when we construct a new `ArrayList` at ❶, we can specify the type of object we want to store in angle brackets: `<Bubble>`.

At ❷, we can use the `setBackground()` method directly because `BubblePanel` is extending `JPanel`, and we've seen that `JPanel` objects have a background color. We're setting the background to the color constant `Color.BLACK` from the `Color` class we imported earlier. This constant has an RGB value of (0, 0, 0).

Now, when we start the BubbleDraw app and create a `BubblePanel` drawing canvas, we'll begin with an empty list of bubbles and a black screen background.

Save your changes so far. In the next section, we'll fill that window with colorful bubbles!

Adding a Method to Draw on the Screen

Next, we need to add a method to paint all the bubbles in `bubbleList` to the drawing screen.

All of the GUI components in the `javax.swing` toolkit, including the `JPanel` we extended for the `BubblePanel` drawing screen, have a `paintComponent()` method that draws that component on the screen. We're going to modify, or *override*, the default `paintComponent()` method to make the `BubblePanel` draw all the bubbles in `bubbleList`.

First, we need to declare the `paintComponent()` method. Because we're

overriding the `paintComponent()` method that already exists in `JPanel`, the parent class of `BubblePanel`, we have to use the same *method signature*, or first line of code. That means we have to match the way `paintComponent()` is declared exactly, as a `public void` method, with one parameter, of type `Graphics`:

```
public class BubblePanel extends JPanel {
    --snip--
    }
    public void paintComponent(Graphics canvas) {
    }
```

We place the `paintComponent()` method just below the `BubblePanel()` constructor. Notice that `paintComponent()` requires a `Graphics` object, which we call `canvas` just as in the `draw()` method we designed for the `Bubble` class. Any object that draws on the screen can use a `Graphics` object to color the individual pixels on the computer screen.

Inside this method, we first want to allow the `JPanel` parent class to clear the drawing screen and do any other setup it normally would before drawing anything else. We do this by calling `paintComponent()` from the parent class:

```
    public void paintComponent(Graphics canvas) {
        super.paintComponent(canvas);
    }
```

The keyword `super` tells Java to call the original `paintComponent()` method from `JPanel`. This means that all the code in the original `paintComponent()` method will be pulled into our new one. This is a useful object-oriented programming feature: because we extended the `JPanel` class to create the new `BubblePanel` class, we can take advantage of all the functionality already built into `JPanel`, such as clearing the pixels inside the window when the app opens and preparing the `Graphics` canvas for drawing colorful graphics. We say that `BubblePanel` *inherits* these functions from `JPanel`.

Once we've prepared the canvas, it's time to loop through the list of bubbles and draw each one on `canvas`. To accomplish this, we'll use a `for` loop in a new way.

You first saw a `for` loop in Chapter 6, when we wanted to loop through each character in a string to encode secret messages. This time, we're using a shorter version of a `for` loop, called a `for each` statement, that's specially designed to loop through a list or collection of objects.

Let's see the working code first, and then I'll break down the `for each` statement piece by piece:

```
public void paintComponent(Graphics canvas) {
    super.paintComponent(canvas);
  ❶ for(Bubble b : bubbleList) {
      ❷ b.draw(canvas);
    }
}
```

You can read the `for each` statement at ❶ as "For each `Bubble b` in `bubbleList`." You can tell this is a `for each` statement instead of a regular `for` loop, first because of the colon in the middle and, second, because it doesn't have the three parts we saw in `for` loops back in Chapter 6: *initialization*, *condition*, and *update*. Java uses the same keyword `for` and parentheses for both loops, but the `for each` statement is designed just for collections of objects, such as arrays, `ArrayLists`, and so on.

For each `Bubble b` in the `ArrayList bubbleList`, the loop will call `b.draw(canvas)` ❷ to draw that individual bubble on the drawing screen. The first time the loop executes, `b` will point to the first `Bubble` object in `bubbleList`, and each time the loop repeats, `b` will point to the next bubble in the list. Calling `b.draw(canvas)` tells that bubble to draw itself on the `canvas`.

This single, short `for each` loop will draw every bubble in the `bubbleList` to the screen. The only problem is that we don't have any bubbles to test. Let's generate some random bubbles to see the app in action before we move on to creating bubbles with the mouse.

Testing the BubblePanel Class

To test what we've done so far, before we add the mouse interaction, we'll write a test method that draws 100 bubbles of random sizes all over the app window. This will allow us to make sure the app works without writing the final code. This way we can debug any errors in our code early and see a preview of how cool the app will look when it's finished.

Call the new method `testBubbles()`. Place it after the `paintComponent()` method and before the `private class Bubble`, as shown in Listing 9-3.

```
public void paintComponent(Graphics canvas) {
    --snip--
```

```
    }
  }
❶ public void testBubbles() {
    ❷ for(int n = 0; n < 100; n++) {
        ❸ int x = rand.nextInt(600);
        ❹ int y = rand.nextInt(400);
        ❺ int size = rand.nextInt(50);
        ❻ bubbleList.add( new Bubble(x, y, size) );
      }
    ❼ repaint();
  }
  private class Bubble {
```

Listing 9-3: Creating the testBubbles() method

At ❶, we declare `testBubbles()` as `public` with `void` as the return type, meaning it doesn't pass any information back to the program. Then, we use a regular `for` loop ❷ to go from `n = 0` to `99`, a total of 100 iterations. That means we'll create 100 bubbles, and for each bubble, we need an (x, y) location and a size for its width and height in pixels.

We set the pixel dimensions of the BubbleDraw app frame to `600` wide by `400` tall at the beginning of this chapter, so for the bubble's location, we need an `x` value between `0` and `600` and a `y` value between `0` and `400`. Inside the `for` loop, we use the random number generator `rand` to get a random integer between `0` and `600` and store it in the variable `x` ❸ for the x-coordinate of the center of the bubble on the screen. Then, we generate a y-coordinate value between `0` and `400` and store it in `y` ❹. Next, for the size of the bubble in pixels, we generate a random number between `0` and `50` ❺. The last step inside the loop creates a new `Bubble` object using the random `x`, `y`, and `size` that we just generated, and we add that new `Bubble` to the `ArrayList` `bubbleList` ❻.

Finally, we call the `repaint()` method ❼. Normally before we draw new computer graphics, we have to clear the screen by drawing an empty, black background, but here `repaint()` does this for us. Notice that we didn't have to clear the background of the screen inside the `paintComponent()` method either; all we had to do was draw the bubbles from `bubbleList`. The `repaint()` method takes care of redrawing the background as well as calling `paintComponent()`, so we'll call it anytime we want to refresh or repaint the screen.

There's just one more step to complete before we can test the app—we need

to call the `testBubbles()` method from the `BubblePanel()` constructor. Add the following line of code to your constructor for the `BubblePanel`:

```
public BubblePanel() {
    bubbleList = new ArrayList<Bubble>();
    setBackground(Color.BLACK);
    testBubbles();
}
```

Save *BubblePanel.java* and then switch to the *BubbleDraw.java* tab. Save that file and press the run button in Eclipse. You'll need to run the file from the *BubbleDraw.java* tab the first time you compile and run, because `BubbleDraw` contains the `main()` method that runs the program. You should see a window full of colorful, randomly placed bubbles, similar to Figure 9-3.

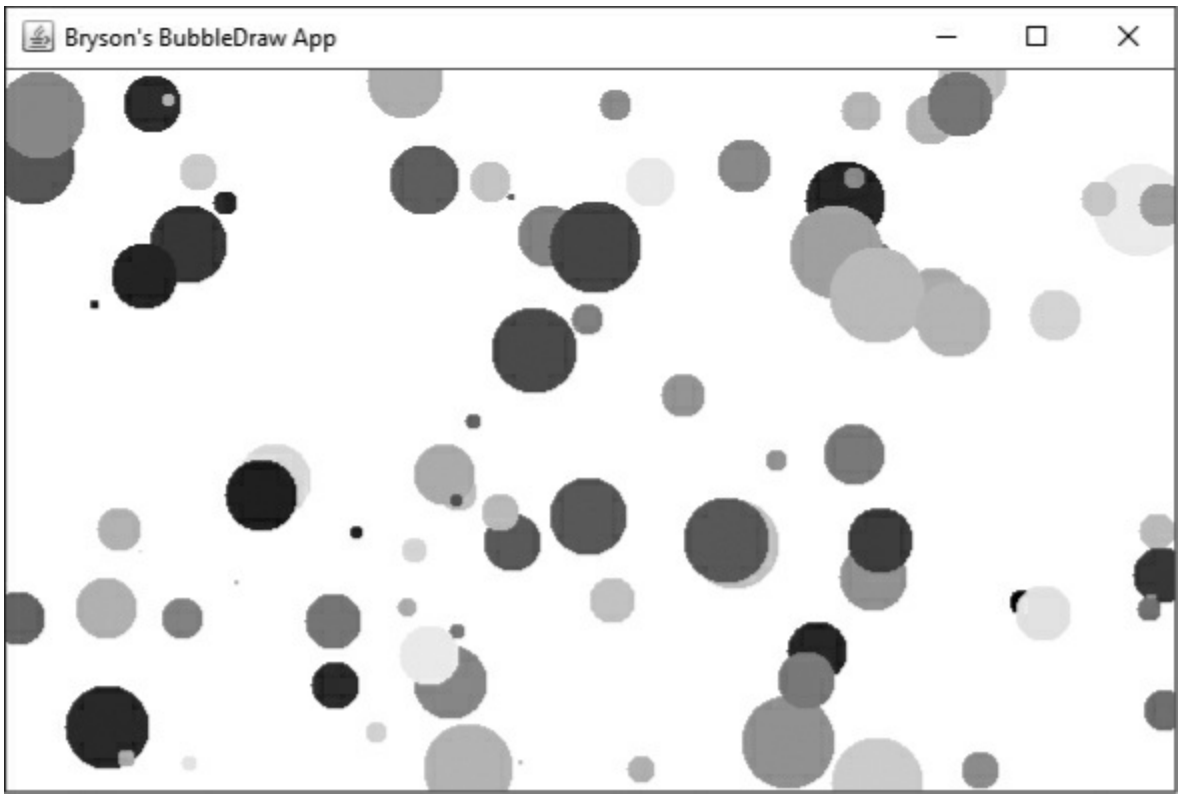


Figure 9-3: Each time you run this version of the BubbleDraw app, the testBubbles() method will draw 100 bubbles on the screen.

If you like the look of the random bubbles, take a little time to play with the numbers in the `testBubbles()` method. See if you can draw 200, 500, or even 1,000 bubbles instead of 100. Make the bubbles larger, either by generating a larger random number for `size` or by adding a value to `size` after the random number is generated. Play with the `x` and `y` values to see if you can fit every

bubble on the screen instead of having some bubbles cut off at the edges.

Modify the `testBubbles()` method as much as you'd like; this is a great opportunity to try new things and immediately see the visual effect of each change. We'll comment out the `testBubbles()` method in the next section as we add mouse interaction, so you can feel free to experiment with `testBubbles()` without fear of messing up the rest of the program.

Handling Mouse Events from the User

The goal of the BubbleDraw app is to allow the user to draw bubbles using their mouse. We saw in the last section that the bubble-drawing part is working. Now all we have to do is add the mouse interaction.

We'll use event listeners to give the app the ability to handle mouse clicks, mouse movement, and even mouse wheel scrolling. We added event listeners to our GUI apps in Chapters 3 and 7 using anonymous inner classes to handle button clicks, sliders, and text field changes; but if we used anonymous inner classes in this app, we would need to make three separate listeners for each type of event, and that would be hard to keep track of. Plus, in two of those events, clicking and dragging the mouse, we want the result to be the same: the app should add bubbles each time the user clicks the mouse on the screen *and* whenever the user drags the mouse across the screen. It would be much easier to just have one block of code we could attach to both events than to write a listener for one event and copy and paste the code to another. So, to make the app more manageable, we're going to create a single, named event listener that can respond to all three types of events.

Creating a Reusable Event Listener

Before we can create our event listener, we need to import one additional library, `java.awt.event.*`, at the top of the *BubblePanel.java* file:

```
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.Graphics;
import java.util.Random;
import java.awt.Color;
import javax.swing.JPanel;
```

Our previous `import` statements have brought in only one class at a time, but this statement brings in every class in the `java.awt.event` library. The wildcard character, the asterisk (`*`), means that we want to bring in all the classes at the top level of `java.awt.event`, including all the mouse events and listeners, and much more. We could import each class individually as we use it, but importing `java.awt.event.*` will make the program much easier to write, allowing us to focus on coding instead of switching back and forth to the top of the file every time we need to use another event class.

Next, let's start coding the private, named inner class to listen for mouse events. We'll name it `BubbleListener`, as it will handle all the events related to bubbles in the `BubblePanel` class. Add this class below your `testBubbles()` method but above the private class `Bubble`. Java programmers usually include listener classes near the bottom of the class file, along with any other helper classes—this convention just helps us quickly find the listener code for debugging or modifying the file.

```
    repaint();
}
private class BubbleListener extends  MouseAdapter {
}
private class Bubble {
```

The `BubbleListener` class extends `MouseAdapter`, which processes mouse events. Just as when we used the `extends` keyword to build a new type of `JFrame` that inherited all the features and functions of its parent class, the `BubbleListener` will inherit all the mouse event listener features of the `MouseAdapter` class. This adapter class includes the ability to handle `MouseListener` events for clicks, `MouseMotionListener` events for mouse movement, and `MouseWheelListener` events for scrolling with the mouse wheel or trackpad.

We'll add these event handlers to the code step-by-step, and we'll test the app after each addition to see the BubbleDraw app emerge one feature at a time.

Handling Clicks and Drags

When we handle mouse events, we need to complete two steps. First, we have to add code to the `BubbleListener` to handle the individual event we're after, like `mousePressed()`. Then, we have to add the listener to the `BubblePanel()` constructor so the drawing screen knows to listen for that type of event and call `BubbleListener` to handle the event when it happens. Let's do this for the first

case, when the user presses the mouse button to draw a bubble.

Listening for Mouse Button Events

There are three events that we can listen for from the mouse buttons:

`mousePressed()` Occurs when the user presses any mouse button

`mouseReleased()` Occurs when the user releases the button

`mouseClicked()` Occurs when the user presses and releases in quick succession

For the BubbleDraw app, we'll use the `mousePressed()` handler so that the app will draw a bubble as soon as the user presses the mouse button. The signature for a `mousePressed()` event handler looks like the following:

```
private class BubbleListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
    }
}
```

NOTE

Spelling and capitalization are especially important when you're writing event handlers because they already have built-in names with specific capitalization. Make sure your mousePressed() method matches this one exactly.

The `mousePressed()` event handler goes inside the `BubbleListener` class, and it must be declared as `public` with a `void` return type to match the `mousePressed()` method in the `MouseAdapter` class. Notice it also takes a parameter of type `MouseEvent`. All of the mouse events get information about the mouse pointer's location on the screen when the event happened. The x- and y-coordinates of the mouse event are stored in a `MouseEvent` object, and we can get those coordinates with the `getX()` and `getY()` methods.

The following code adds a bubble to the `bubbleList` at the location where the user clicked the mouse, and then repaints the drawing screen to make the bubble appear.

```
private class BubbleListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      ❶ bubbleList.add(new Bubble(e.getX(), e.getY(), size));
```

```
            ❷ repaint();
        }
    }
```

At ❶, we create a new `Bubble` object at the (x, y) location where the user clicked the mouse, `e.getX()` and `e.getY()`. The `MouseEvent` class has several properties and methods for working with mouse events, like finding out which button was pressed and where the mouse is on the screen. As I just mentioned, the `getX()` and `getY()` methods tell us the x- and y-coordinates of the mouse event, such as a click or drag. Look back down in the `Bubble` class, and you'll see that the constructor method we wrote there required three parameters: `int newX`, `int newY`, and `int newSize`. So, we must pass `e.getX()`, `e.getY()`, and `size` to the constructor to create a new `Bubble` object. Once the bubble is created, we add it to the `ArrayList` of bubbles with `bubbleList.add()`.

At ❷, we call the `repaint()` method to refresh the screen and draw the updated `bubbleList` to the canvas.

This completes the `mousePressed()` event handler, but we still need to do one more thing to make the app listen for `mousePressed()` events and send them to the `BubbleListener` class. We have to tell the `BubblePanel` class to add `BubbleListener` as the listener for mouse events.

In the `BubblePanel()` constructor, make two changes. First, comment out the `testBubbles()` line by placing two forward slashes at the front of the line. Second, call the `addMouseListener()` command to use `BubbleListener` to process mouse events. The updated constructor is shown here:

```
    public BubblePanel() {
        bubbleList = new ArrayList<Bubble>();
        setBackground(Color.BLACK);
      ❶ // testBubbles();
      ❷ addMouseListener( new BubbleListener() );
    }
```

By commenting out the call to `testBubbles()`❶, we've left it in place in case we want to draw random test bubbles again, but we've stopped it from running so that we can test the interactive BubbleDraw app using the mouse. The new line at ❷ tells the `BubblePanel` to listen for mouse events and to send them to the `BubbleListener` class when they happen.

With those changes in place, you should be able to run the BubbleDraw app

and use the mouse button to place bubbles anywhere you click, as shown in Figure 9-4.

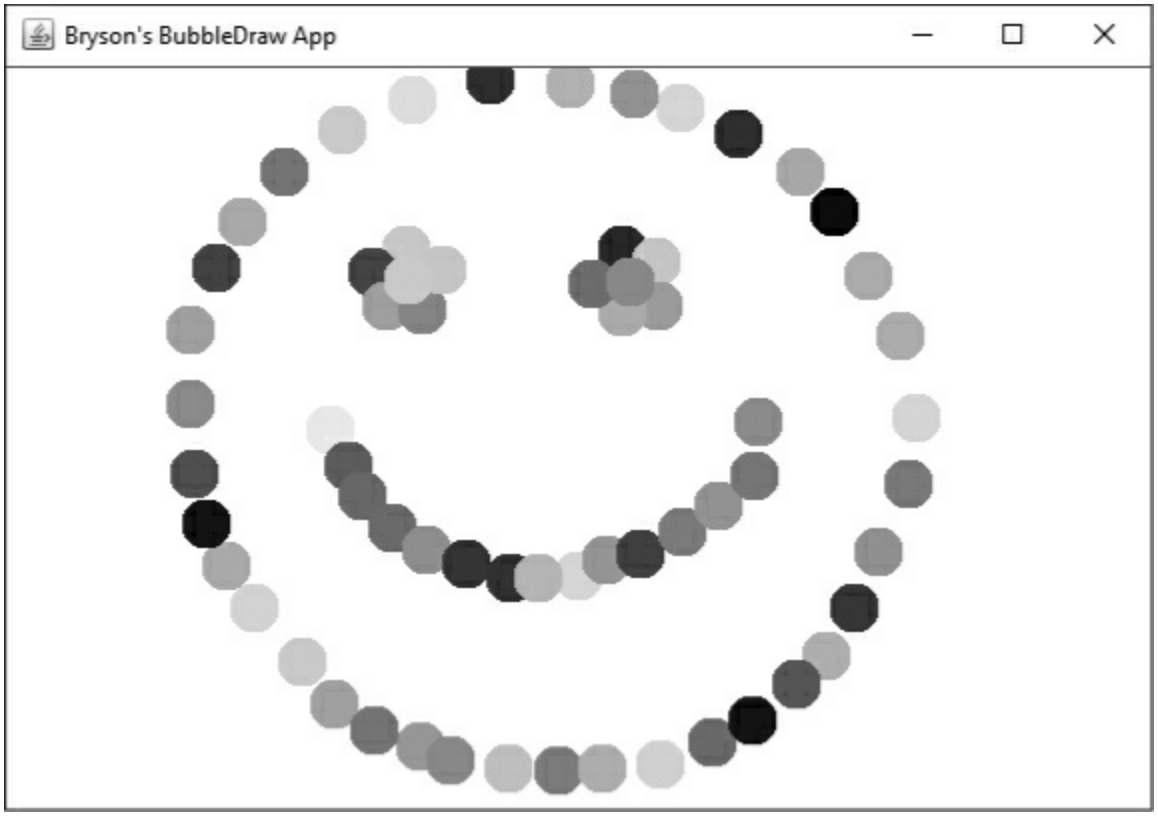


Figure 9-4: After adding the mousePressed() handler and setting BubbleListener as the listener for mouse events in the app, you should be able to draw by clicking anywhere on the screen.

Very cool! You can click repeatedly to draw shapes and patterns on the screen, but it would be easier to drag the mouse to draw bubbles. Let's address that next.

Listening for Mouse Motion Events

Mouse motion events are a separate type of event from mouse button presses, but they can still be processed in a class that extends `MouseAdapter`, like `BubbleListener`. The two types of mouse motion events are `mouseMoved()` and `mouseDragged()`.

A `mouseMoved()` event happens anytime the mouse is moved over the drawing window. A `mouseDragged()` event occurs whenever the mouse is moved while a button is pressed. Since we want to draw bubbles only when the user clicks and drags the mouse, we'll implement the `mouseDragged()` event handler in the `BubbleListener` class first, and then we'll add a mouse motion listener to the

`BubblePanel` constructor to activate the listener.

Add the `mouseDragged()` event handler to the `BubbleListener` class, as shown here:

```
  private class BubbleListener extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
          --snip--
      }
      public void mouseDragged(MouseEvent e) {
          bubbleList.add(new Bubble(e.getX(), e.getY(), size));
          repaint();
      }
  }
```

You'll notice that the code looks almost exactly like the `mousePressed()` event handler, except for the name `mouseDragged()`. That's because they both handle `MouseEvents`. The `mousePressed()` method handles events when the user presses a mouse button, and `mouseDragged()` is called whenever the user drags the mouse. In this app, both events should behave the same way, adding bubbles to `bubbleList` and calling the `repaint()` function.

Next, add `BubbleListener` as the mouse motion listener to the `BubblePanel()` constructor with the following statement:

```
    public BubblePanel() {
        bubbleList = new ArrayList<Bubble>();
        setBackground(Color.BLACK);
        // testBubbles();
        addMouseListener( new BubbleListener() );
        addMouseMotionListener( new BubbleListener() );
    }
```

Once again, let's run the app to test this new event listener. Save your code, press the run button, and click and drag around the app window that appears, as shown in Figure 9-5.

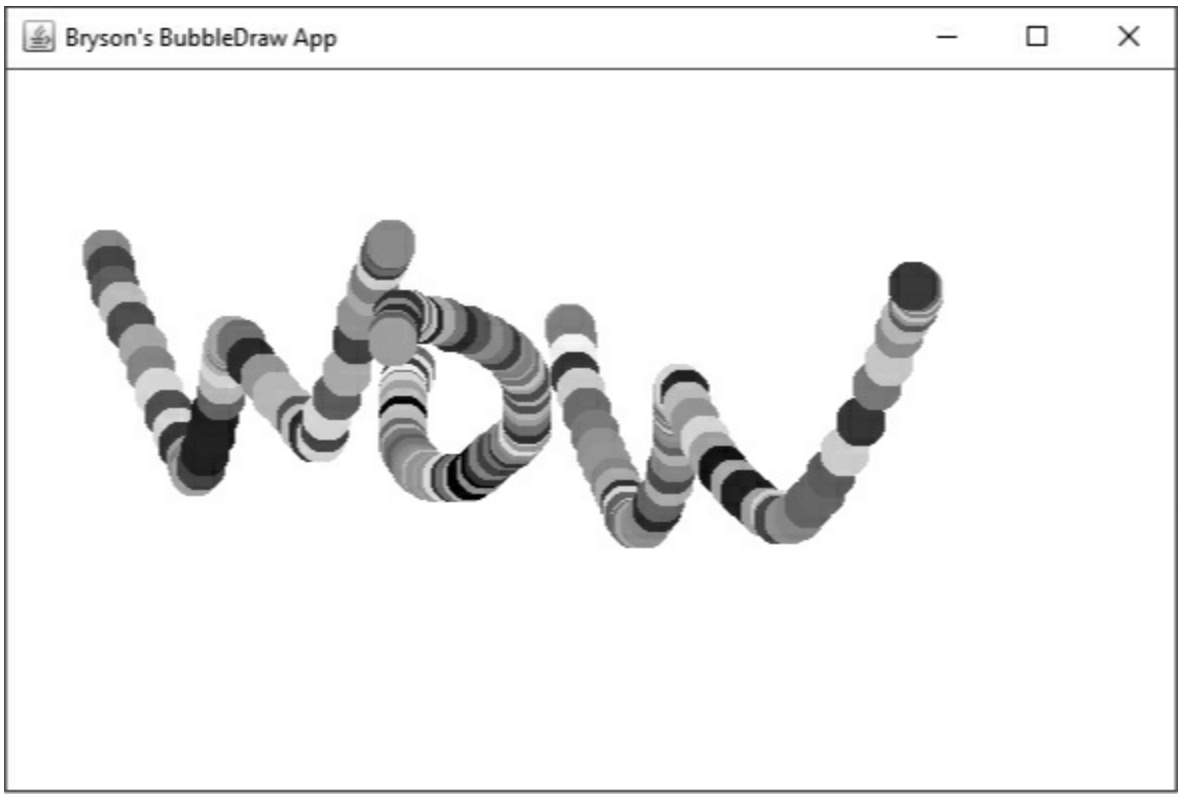


Figure 9-5: Now you can click and drag to draw continuous streams of bubbles!

Impressive, huh? Just by adding two event handlers for mouse button and mouse motion events to the `BubblePanel` graphics app, we've built an interactive, multicolored drawing app in Java.

Bonus: Handling MouseWheel Events

We've seen two small but powerful event handlers for mouse button presses and mouse click-and-drag events at work in the app so far, but there's another event class in `MouseAdapter`: `MouseWheelEvent`. Depending on your system, you might have a physical wheel on your mouse or trackpad that you can use to scroll up and down in documents and web pages. Or, you might have a laptop with a trackpad that allows you to scroll using a gesture, such as the two-finger swipe gesture on a MacBook or the vertical swipe along the far-right edge of the trackpad on most Windows laptops.

Java interprets either kind of scroll event as a `MouseWheelEvent`. We can add a `mouseWheelMoved()` listener to the `BubbleListener` to handle mouse wheel events or trackpad scroll gesture events on systems that provide them, as follows:

```
    private class BubbleListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
```

```
            --snip--
        }
        public void mouseDragged(MouseEvent e) {
            --snip--
        }
    ❶ public void mouseWheelMoved(MouseWheelEvent e) {
      ❷ size += e.getUnitsToScroll();
        }
}
```

At ❶, the `mouseWheelMoved()` method accepts a `MouseWheelEvent`. At ❷, we get the number of units the mouse wheel scrolled from the `MouseWheelEvent e` with `e.getUnitsToScroll()`, and we add this amount to `size`. The number returned by `getUnitsToScroll()` can be positive or negative, depending on your operating system and whether the wheel was scrolled up or down.

DIFFERENCES IN SCROLLING BETWEEN OPERATING SYSTEMS

Windows, macOS, and Linux differ when it comes to scrolling. On macOS, scrolling or gesturing up returns a positive value, so the code we just wrote will increase the bubble size when the user scrolls or gestures up and shrink the size when the user scrolls down. On Windows and most Linux flavors, scrolling up with the mouse wheel moves a web page or document *down* instead, returning a negative number from `getUnitsToScroll()`. This means that scrolling up will cause the bubbles to get smaller, while scrolling down will add to the size and make them larger.

If you want to make your app work the same way on all three operating systems, change the code to the following:

```
        public void mouseWheelMoved(MouseWheelEvent e) {
            if(System.getProperty("os.name").startsWith("Mac"))
                size += e.getUnitsToScroll();
            else
                size -= e.getUnitsToScroll();
        }
```

The `if` statement gets the operating system name with `System.getProperty("os.name")` and checks whether the name starts with the string `"Mac"`. If so, we add the units scrolled to `size`. Otherwise, we subtract the units scrolled from `size` to reverse Windows and Linux scrolling to positive for up,

negative for down.

Lastly, add the mouse wheel listener to the `BubblePanel()` constructor, just below the previous two listeners:

```
    public BubblePanel() {
        bubbleList = new ArrayList<Bubble>();
        setBackground(Color.BLACK);
        // testBubbles();
        addMouseListener( new BubbleListener() );
        addMouseMotionListener( new BubbleListener() );
        addMouseWheelListener( new BubbleListener() );
    }
```

Save and run the app now, and you should be able to change the bubble size by scrolling your mouse wheel or gesturing up or down on your trackpad, as shown in Figure 9-6.

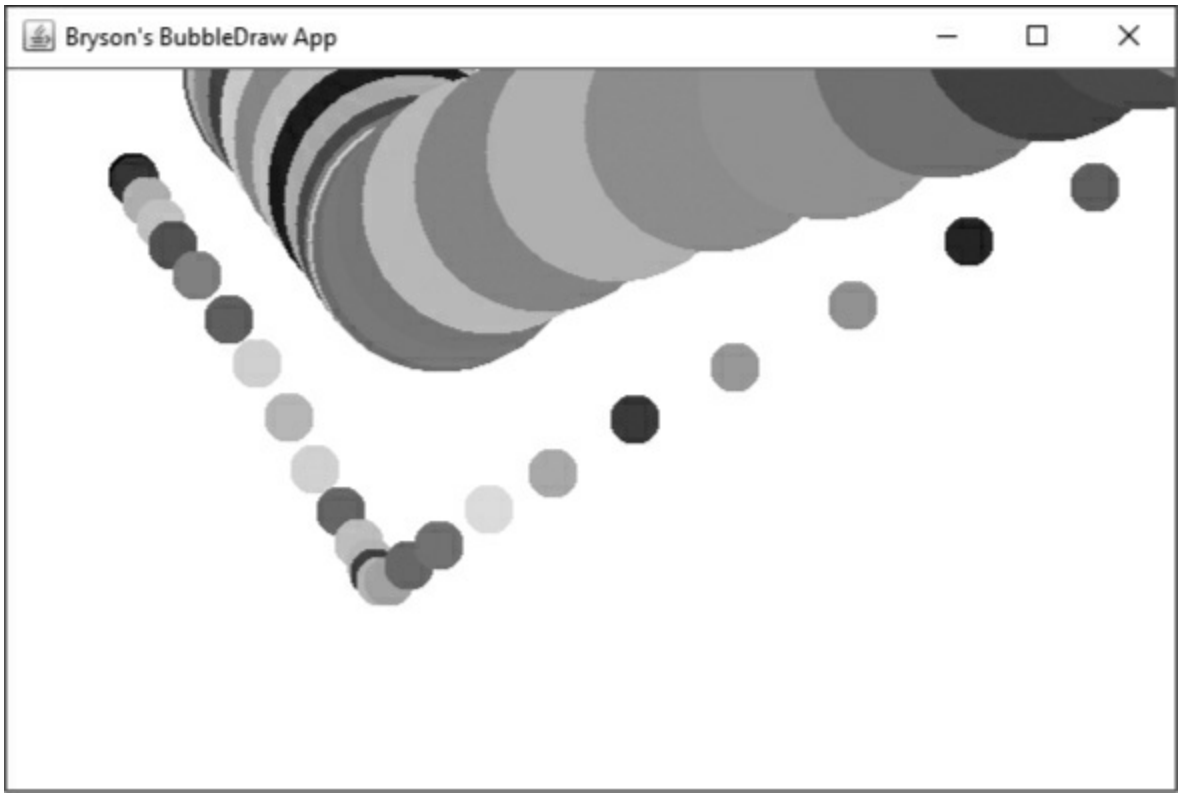


Figure 9-6: Use your mouse wheel or trackpad to change the size of the bubbles.

The first version of the BubbleDraw app is complete! You can click and drag to draw with multicolored bubbles on the screen, scroll up and down to change the bubble size, maximize the window to draw fullscreen, and more.

What You Learned

You've designed a colorful, interactive drawing app though an object-oriented programming approach by creating a `Bubble` class to capture the features and behaviors of bubbles.

Here are some of the new things you've learned in this chapter:

- Building a single app with multiple classes and multiple Java source code files
- Using an object-oriented programming approach to break a problem down into smaller pieces
- Building the `Bubble` class from scratch, including attributes, methods, and a constructor
- Creating random colors using the `java.util.Random` and `java.awt.Color` classes
- Generating random numbers in a given range with `Random.nextInt(range)`
- Drawing graphics with the `java.awt.Graphics` class
- Using the `paintComponent()` method to describe what your app should draw and using `repaint()` to clear the screen and redraw
- Writing event handlers for mouse events, including clicks, movement, and mouse wheel events
- Using the `java.util.ArrayList` class to store a dynamic list of objects, such as the bubbles in BubbleDraw
- Adding multiple event listeners to an app using a custom listener class, `BubbleListener`

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned, as well as to expand your programming skills. Visit the book's website at *https://www.nostarch.com/learnjava/* for sample solutions.

#1: No Bubble Too Small

If you scroll the mouse down enough, the bubbles will simply disappear because

the `size` value drops below `0`, resulting in a circle that Java can't draw on the screen. In fact, a `size` of `2` or smaller will produce bubbles just a single pixel tall, so let's make sure the value of `size` stays at least 3 pixels by 3 pixels to keep the bubbles circular.

Add a little logic to the `mouseWheelMoved()` method in the `BubbleListener` class in *BubblePanel.java* to check whether the `size` variable goes below a value of `3`. If so, set `size` equal to `3` to keep the bubble visible on the screen. Add this new `if` statement to the end of the `mouseWheelMoved()` method so that you check the size after it's changed, not before.

Run the app with the change, and you should be able to scroll the mouse wheel until the bubbles are just a couple of pixels wide, but they'll no longer disappear. Great work!

#2: PixelDraw!

Add just a little bit of math to the constructor for new `Bubble()` objects, and you can create a cool, pixelated effect, as shown in Figure 9-7.

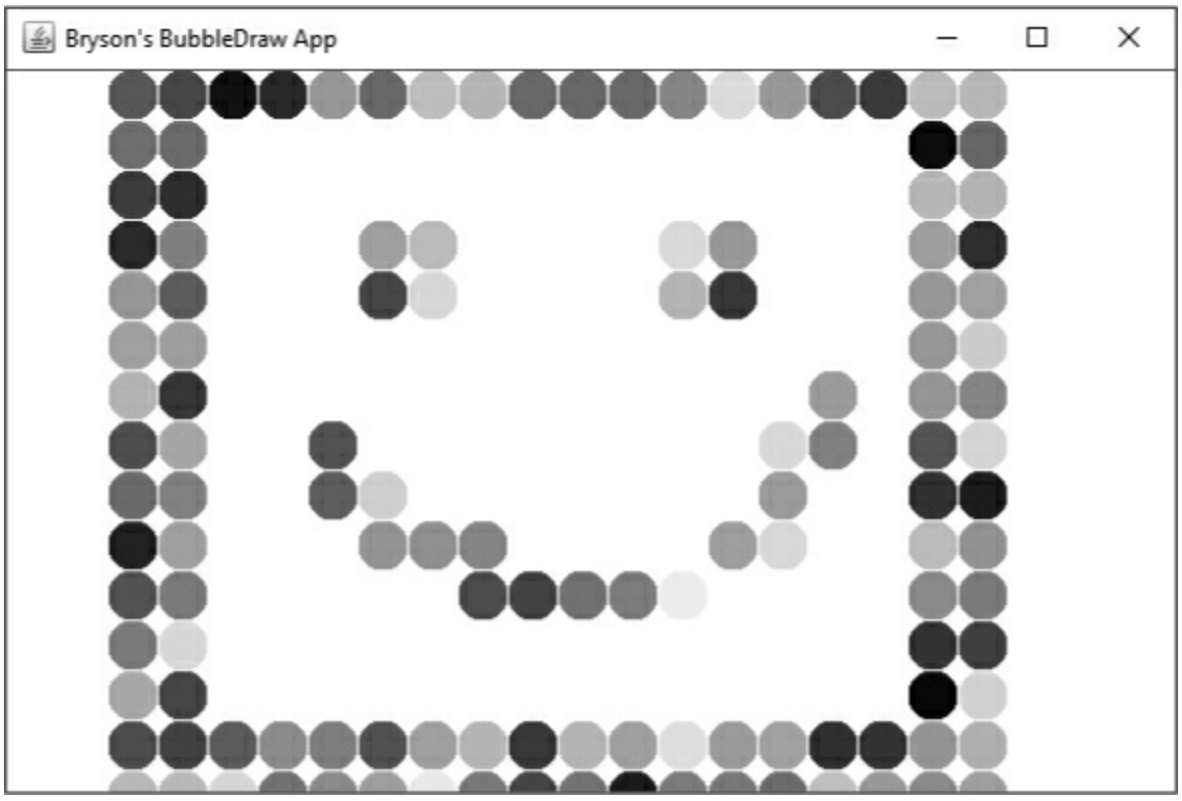


Figure 9-7: Changing the location of new Bubbles slightly can create a cool pixel-drawing effect.

Let's copy the *BubbleDraw* project folder to make the changes in a new project

without messing up the original app. Double-click your *BubblePanel .java* tab to restore the default view and then, in the Package Explorer, press CTRL-C or ⌘-C to copy the *BubbleDraw* project folder. Press CTRL-V or ⌘-V to paste a new copy of the folder into your workspace. Change the name of the folder to **PixelDraw**, as shown in Figure 9-8.

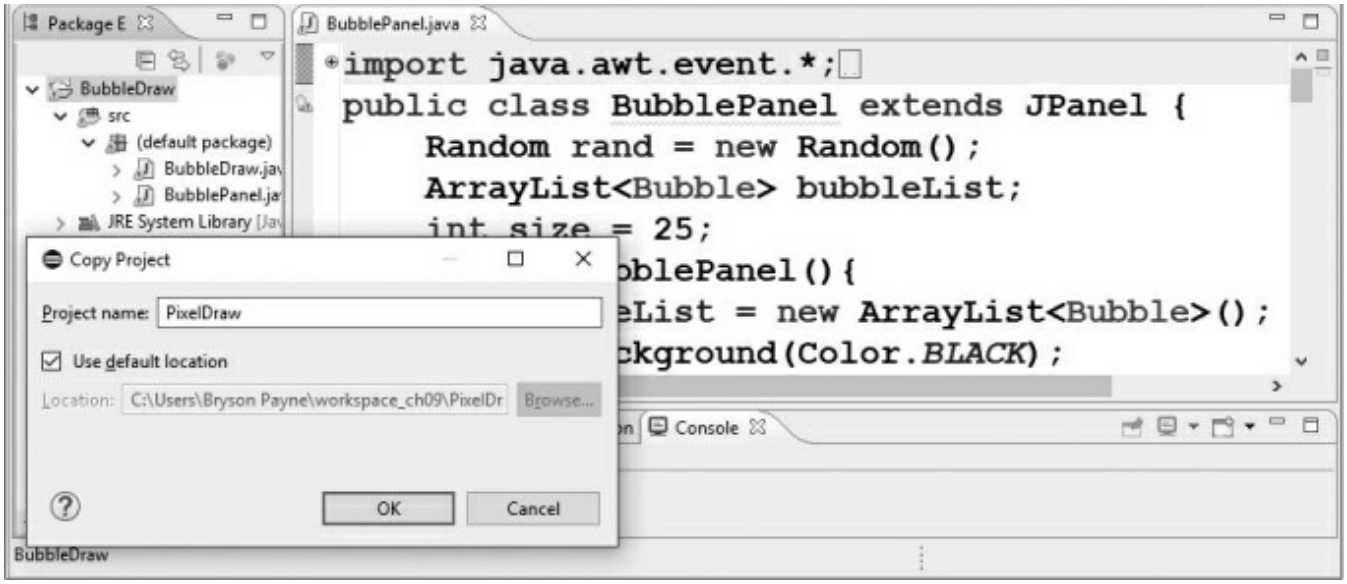


Figure 9-8: You can copy a project folder in the Package Explorer and paste it to create a new copy.

Change the code for the `Bubble()` constructor inside the private class `Bubble` as follows.

```
    public Bubble(int newX, int newY, int newSize) {
        x = (newX / newSize) * newSize + newSize/2;
        y = (newY / newSize) * newSize + newSize/2;
        size = newSize;
        color = new Color( rand.nextInt(256),
                rand.nextInt(256),
                rand.nextInt(256) );
    }
```

This new math only alters the (x, y) location of each new bubble because of the way Java handles division of two integer values. Java keeps only the integer, or whole number, result when you divide two integers, so dividing by `newSize` and multiplying by `newSize` will make your bubbles appear to be on a grid, as you saw back in Figure 9-7. For example, if `newSize` is `10` and `x` is `25`, then `x` divided by `newSize` would be `25 / 10`, which evaluates to `2` in Java. Multiplying by `10` then gives us `20`, the location of the bubble on a 10×10 pixel grid. We add `newSize/2` to each coordinate to center the dots within the imaginary grid. Remember,

scrolling the mouse wheel still changes the size of the bubbles, so you can draw thick, blocky images or fine, pixel-perfect drawings.

If you'd like to make the pixel effect more *Minecraft*-y, you can make the bubbles into squares. To do so, change the `draw()` method to fill rectangles instead of ovals by commenting out the `fillOval()` command and using the `fillRect()` command instead:

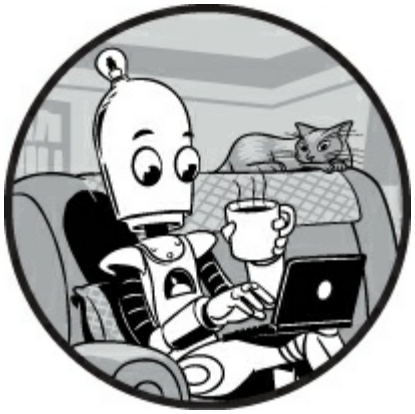
```
    public void draw(Graphics canvas) {
        canvas.setColor(color);
        // canvas.fillOval(x - size/2, y - size/2, size, size);
        canvas.fillRect(x - size/2, y - size/2, size, size);
    }
```

Now you can draw in a pixelated style, as shown in Figure 9-9.

Figure 9-9: Drawing with rectangles instead of ovals creates a pixelated drawing effect.

10

ADDING ANIMATION AND COLLISION DETECTION WITH TIMERS



In this chapter, we'll add timer-based animation to our BubbleDraw app to create floating, bouncing bubbles, and we'll enhance the app with a user-friendly GUI interface. The enhanced app, called BubbleDrawGUI, will add a `JPanel` containing the GUI components shown in Figure 10-1, giving the user the power to animate or pause the bubbles, change the animation speed, and clear the screen.

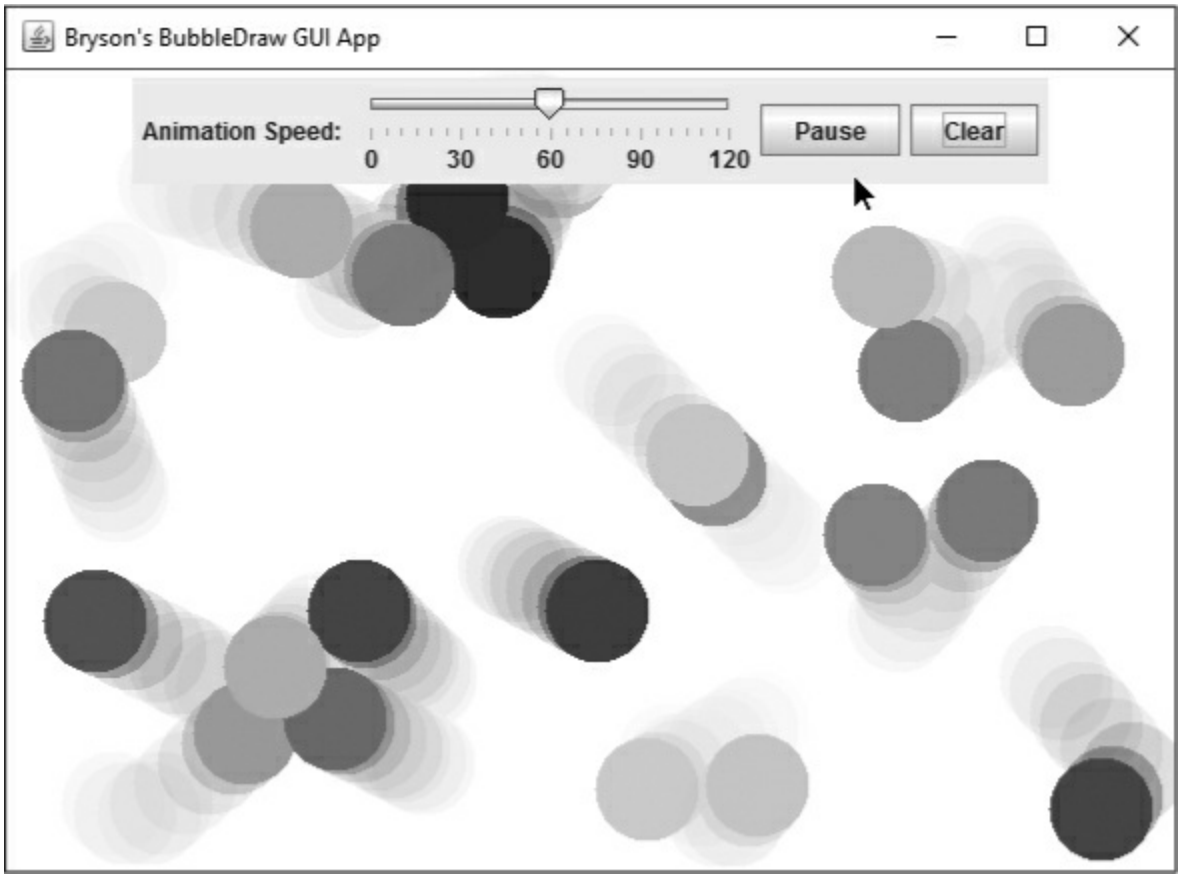


Figure 10-1: The enhanced BubbleDrawGUI app features animated, bouncing, semitransparent bubbles, with a GUI interface to control the animation.

This version of the app is even more interactive and user-friendly than before, allowing the user to draw with floating, bouncy bubbles as they click and drag their mouse on the screen.

Copying the BubbleDraw Java Project to Create BubbleDrawGUI

The new GUI app will build directly on the BubbleDraw project from Chapter 9. So, instead of creating a new Java project from scratch, we'll copy the BubbleDraw project and paste it into the same workspace with a new name. This is a useful approach anytime you want to expand and work on a newer version of a program while keeping the previous version intact.

In Eclipse, right-click the *BubbleDraw* project folder in the Package Explorer pane and select **Copy**. Then, right-click in the Package Explorer pane and select **Paste**. The Copy Project pop-up window will allow you to give the copied

project a new name. Enter **`BubbleDrawGUI`** and click **OK**. Eclipse will create a new copy of the BubbleDraw project in the Package Explorer pane as BubbleDrawGUI.

Renaming the Main Class and Java File

Now let's rename the *BubbleDraw.java* file. This is the file with a `public static void main()` method that runs the application, and renaming it will help us distinguish between the new app and the old version. Inside the new *BubbleDrawGUI* project folder, right-click *BubbleDraw.java* and select **Refactor ▸ Rename**.

Refactoring means restructuring your code, but not its functionality. Usually programmers refactor when they've thought of a better, more efficient way of getting their code to do the same thing. When the Rename Compilation Unit window pops up, enter the new name, **`BubbleDrawGUI`**, and then click **Finish**. A second window may pop up, warning you that the class contains a `main()` method. You can ignore this warning and just click **Finish** a second time. The refactoring process will rename both the class and the Java file to BubbleDrawGUI. We'll leave the `BubblePanel` class unchanged for now.

Finally, let's modify the `JFrame` window's title to match the new GUI version of the app. Open the *BubbleDrawGUI.java* file. Find the line that creates the `JFrame` and modify it to say *Your Name's* `BubbleDraw GUI App` as follows:

```
import javax.swing.JFrame;
public class BubbleDrawGUI extends JFrame {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Your Name's BubbleDraw GUI App");
```

NOTE

The first time you run an app containing multiple files, such as the BubblePanel and BubbleDrawGUI apps, you'll need to run the file that contains the main() method. Running the main file will create configurations that enable you to run the program just by clicking the run button from then on. The BubblePanel class doesn't contain a main() method, so we must either run BubbleDrawGUI.java *or right-click the* BubbleDrawGUI *project folder and select* ***Run As ▸ Java Application****.*

Save your file and then run it to see the new title in the title bar at the top of the window, as shown in Figure 10-2.

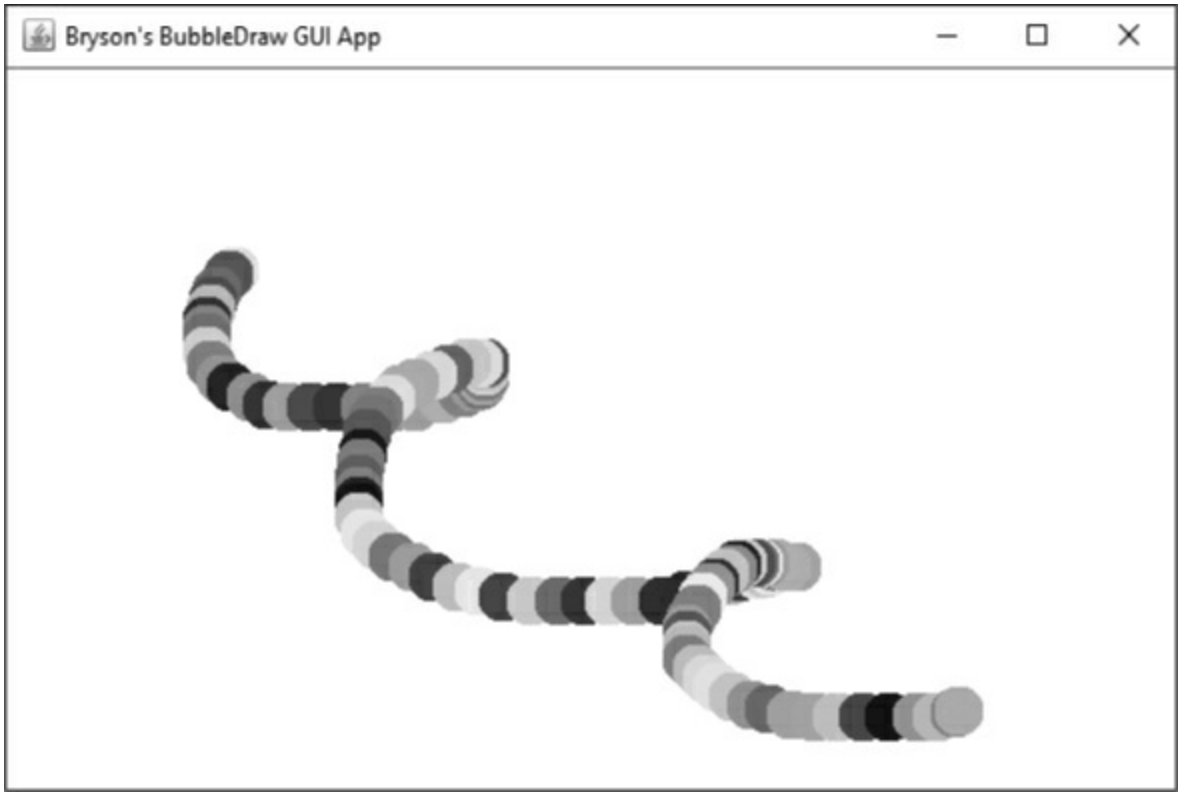


Figure 10-2: The new BubbleDrawGUI.java *file opens a window with* "Your Name's *BubbleDraw GUI App" in the title bar.*

Now, let's make one more change to make the bubbles look more realistic.

Adding Transparency

Real bubbles often have a semitransparent look. Think about blowing a bubble with bubble gum: once it gets big enough, you can see through the thin surface of the bubble. We can add transparency to the bubbles in the BubbleDrawGUI app to give them a more lifelike appearance.

In addition to the RGB color components you learned about in Chapter 9, Java can store a fourth component in the `java.awt.Color` class. This is called the *alpha* component, and it represents how transparent or opaque a color should appear onscreen when it is drawn in front of other objects. Just like the RGB color values, the alpha component can have values from `0` to `255`. An alpha value of `0` will make the color invisible, `128` will make it look semitransparent like watercolor paint, and `255` will make it completely obscure any objects behind it.

Because the `Color` class's constructor can accept the alpha value as a fourth argument (right after the RGB color values), we only need to change one line in the *BubblePanel.java* file to add transparency. Open *BubblePanel.java* under the BubbleDrawGUI project's *src* folder and scroll to the bottom of the file where the `Bubble` class is defined:

```
private class Bubble {
        private int x;
        --snip--
            color = new Color(rand.nextInt(256),
                    rand.nextInt(256),
                    rand.nextInt(256),
                    rand.nextInt(256) );
        }
```

Here we change the color variable's constructor by adding a fourth random value that can range from `0` to `255`. We do so by adding a comma at the end of the third `rand.nextInt(256)` in the `color` statement and adding a fourth `rand.nextInt(256)` before the closing parenthesis of the `Color()` constructor. Be careful to check your commas and parentheses against the code shown here, or the app won't work.

Save the file and then run it. Click around the screen to draw dots that overlap slightly, as shown in Figure 10-3.

You'll see now that the bubbles have not only random colors but also varying levels of transparency. Some bubbles are opaque and completely cover the screen behind them, while others are so transparent they're barely visible. Our bubbles are more bubbly than ever! Now let's make them float for an even more realistic appearance.

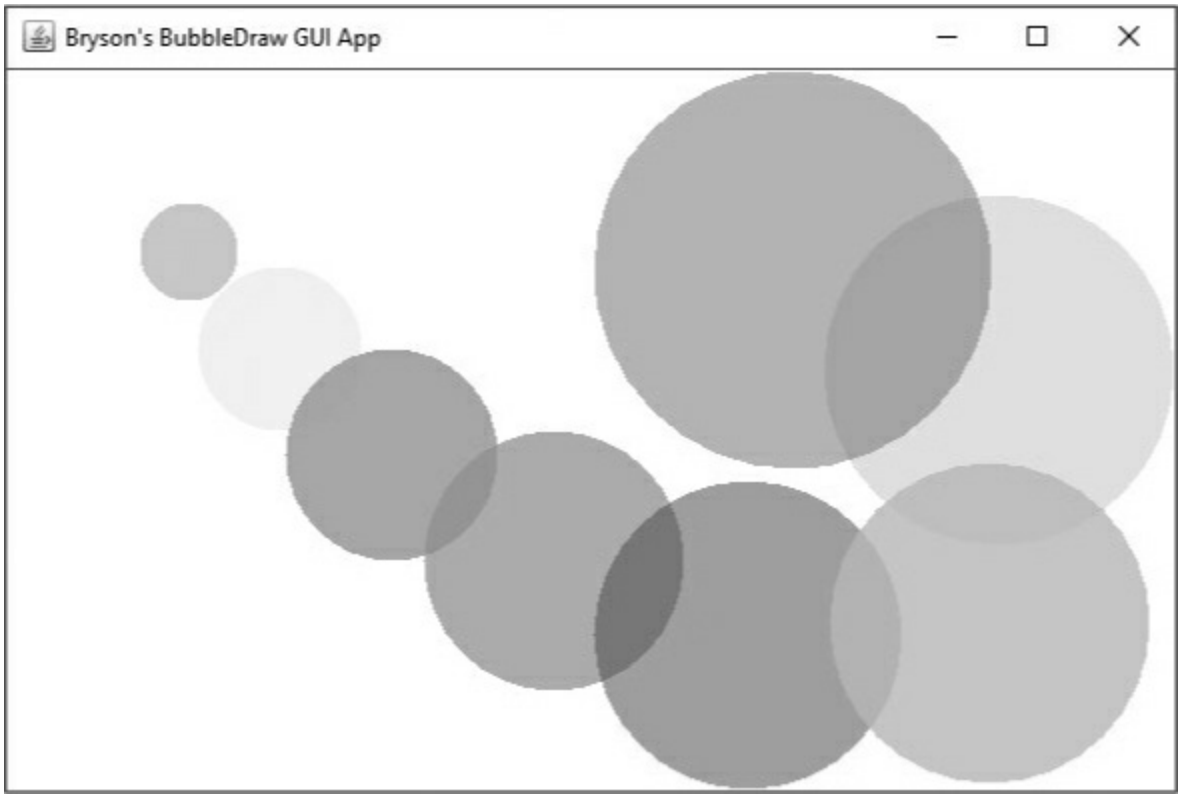


Figure 10-3: Adding an alpha component to the color of each bubble gives them a cool, transparent appearance.

Adding Animation: Bubbles Rising!

Animation is the illusion of movement created by showing a sequence of images on the screen. You may have created a flipbook-style animation in a notebook: each drawing was shifted slightly from the previous one, and when you flipped through the notebook, you could see the animation come to life. We'll add this type of effect to make it look like the bubbles are floating away in the BubbleDrawGUI app.

To animate the bubbles, we need to draw all the bubbles on the screen, change their location slightly, and then draw the screen again a few times every second. Each screen we draw is called a *frame*. If we redraw the objects quickly enough, our eyes and brains fill in the gaps between frames, making us believe the same object has moved in a smooth path. The animation's *frame rate*, or how quickly we redraw the screen, is usually around 30 frames per second. We'll use a new class, `javax.swing.Timer`, which creates timers, to tell our program when to redraw the bubbles. We'll also use an event handler to update the bubbles' location and repaint the screen each time the timer goes off.

There are four steps to create the animated bubbles: adding a timer, setting the timer, preparing the animation, and starting the timer. These are the same steps you would use to add animation to a game or any other app that uses a timer in Java.

Adding a Timer

To add a timer to our app, we'll need to import the `javax.swing.Timer` class. At the top of your *BubblePanel.java* file, add the following `import` statement:

```
import javax.swing.Timer;
import java.awt.event.*;
import java.util.ArrayList;
import java.awt.Graphics;
import java.util.Random;
import java.awt.Color;
import javax.swing.JPanel;
```

Importing the `Timer` class from `javax.swing` enables us to create a `timer` object that triggers an event as often as we choose. Notice on the second line in the code snippet, we have already imported `java.awt.event.*`. This line imports all the `java.awt` event handlers, including the `ActionListener` class we'll use to handle timer events.

Next, inside the `BubblePanel` class, add two variables: one named `timer` for the timer itself and an `int` named `delay` to store how many milliseconds the timer should wait before redrawing the screen:

```
public class BubblePanel extends JPanel {
    Random rand = new Random();
    ArrayList<Bubble> bubbleList;
    int size = 25;
    Timer timer;
    int delay = 33;
```

Timers in Java need to know how many milliseconds, or thousandths of a second, to wait until they trigger a timer event. A millisecond is really fast, so I've chosen a delay of 33 milliseconds. This will cause the screen to be redrawn about 30 times per second, since 1 second = 1,000 milliseconds and 1,000 / 33 = 30 drawings per second. This is about the same rate as a cartoon on television.

Setting the Timer

Now we're ready to set the timer. Inside the `BubblePanel()` constructor, add the following line to initialize the timer and set it with the given delay:

```
   public BubblePanel() {
       timer = new Timer(delay, new BubbleListener() );
       bubbleList = new ArrayList<Bubble>();
       setBackground(Color.BLACK);
       // testBubbles();
       addMouseListener( new BubbleListener() );
       addMouseMotionListener( new BubbleListener() );
       addMouseWheelListener( new BubbleListener() );
   }
```

The constructor for a new `Timer()` requires two parameters: the first is the delay in milliseconds, and the second is the event handler that will listen for timer events. A timer triggers an `actionPerformed()` event every time it goes off, similar to the `actionPerformed()` event we handle for button clicks in a GUI interface. A timer is sort of like an automatic button that "clicks" itself every few milliseconds. We've placed the timer first in the constructor so that we can change it in response to GUI events later.

To listen for timer events, we're going to modify the `BubbleListener` class. Scroll down in *BubblePanel.java* to find the private class `BubbleListener` that we created earlier. Then add `implements ActionListener` before the opening brace for the class:

```
private class BubbleListener extends MouseAdapter implements ActionListener {
```

This change allows the `BubbleListener` class to listen for `actionPerformed()` events by implementing the `ActionListener` class from `java.awt.event.*`. Implementing an event listener class is another way of handling user events. To handle these timer events, we'll need to add an `actionPerformed()` event handler. Add the following method to the bottom of the `BubbleListener` class:

```
   private class BubbleListener extends MouseAdapter implements ActionListener
{
       public void mousePressed(MouseEvent e) {
           --snip--
       }
       public void mouseDragged(MouseEvent e) {
           --snip--
       }
       public void mouseWheelMoved(MouseWheelEvent e) {
           --snip--
       }
```

```
    public void actionPerformed(ActionEvent e) {
    }
}
```

This new `actionPerformed()` method is where we'll add the code that moves the bubbles and repaints the screen every time the timer goes off. We'll add those statements next.

Preparing the Animation

Now that we've added a timer and set up the `BubbleListener` to listen for timer events, we need to tell Java what to do when the timer triggers an event.

Every time the timer fires off an event, it's time to draw the next image in the animated bubble sequence. First, we'll tell the `actionPerformed()` event handler to update the bubbles and redraw the screen. Then, we'll tell the `Bubble` class to update a bubble by moving it upward on the screen. We'll have the program do these steps about 30 times per second.

Inside the `actionPerformed()` method that we added to the `BubbleListener` class, add the following three lines of code to update the bubbles and repaint the screen:

```
    public void actionPerformed(ActionEvent e) {
     ❶ for (Bubble b : bubbleList)
         ❷ b.update();
     ❸ repaint();
    }
```

At ❶, we're using the `for-each` version of a `for` statement to loop through each `Bubble b` in `bubbleList`. Remember that `bubbleList` is the `ArrayList` containing all the bubbles we've created by clicking and dragging the mouse on the screen.

As we loop through each bubble in `bubbleList`, we call a new function named `update()` ❷ on all the bubbles. Eclipse underlines this statement in red because we haven't yet defined the `update()` method in the `Bubble` class, but we'll do that in just a moment. The `update()` method is where we'll change the location of the bubbles to make them look like they're floating up toward the top of the screen.

At ❸, we call the `repaint()` method to refresh the screen, clearing the drawing window and painting the bubbles in their new, updated locations. By doing this 30 times per second, we'll achieve the animated effect we want to see.

Now let's create the `update()` method to tell the `Bubble` class how to move bubbles each time the animation timer is triggered. In Java's (x, y) coordinate system, we need to subtract from the `y` value (the top of the screen is where `y` equals `0`). So, to make the bubbles appear to move upward, we can subtract a small amount from the y-coordinate at each update.

Scroll down to the bottom of *BubblePanel.java* where we defined the `Bubble` class and add the `update()` method just below the `draw()` method, as shown here:

```
        public void draw(Graphics canvas) {
            canvas.setColor(color);
            canvas.fillOval(x - size/2, y - size/2, size, size);
        }
        public void update() {
            y -=5;
        }
    }
}
```

The function to update the position of a bubble subtracts five pixels from the `y` value for that bubble. Each time a bubble is redrawn on the screen, it will be located five pixels higher than before.

Save your file. You only have one more step before you can run the app!

Starting the Timer

The final step in animating the BubbleDrawGUI app is starting the timer. Scroll up to the `BubblePanel()` constructor and add the following line:

```
    public BubblePanel() {
        timer = new Timer(delay, new BubbleListener() );
        --snip--
        addMouseWheelListener( new BubbleListener() );
        timer.start();
    }
```

The `timer.start()` method will start the timer so that it will fire events every few milliseconds as specified, until the `timer.stop()` method is called or until you exit the program.

Save and run the program now. When you draw bubbles, they should float upward, smoothly animated by the timer event handler.

The mouse scroll wheel and every other feature we built in Chapter 9 still work with our mesmerizing animation effect. The bubbles float in only one

direction so far, but we've achieved the illusion of motion we were aiming for. In the next section, you'll learn how to make the bubbles float in every direction.

Forever Blowing Bubbles: Adding Random Speed and Direction

The `update()` function we created in the previous section changed only the y-location of each bubble, causing the bubbles to move vertically every time the screen was redrawn. In this section, we'll make the bubbles move both vertically and horizontally at random speeds so they appear to be blowing away in every direction, as shown in Figure 10-4.

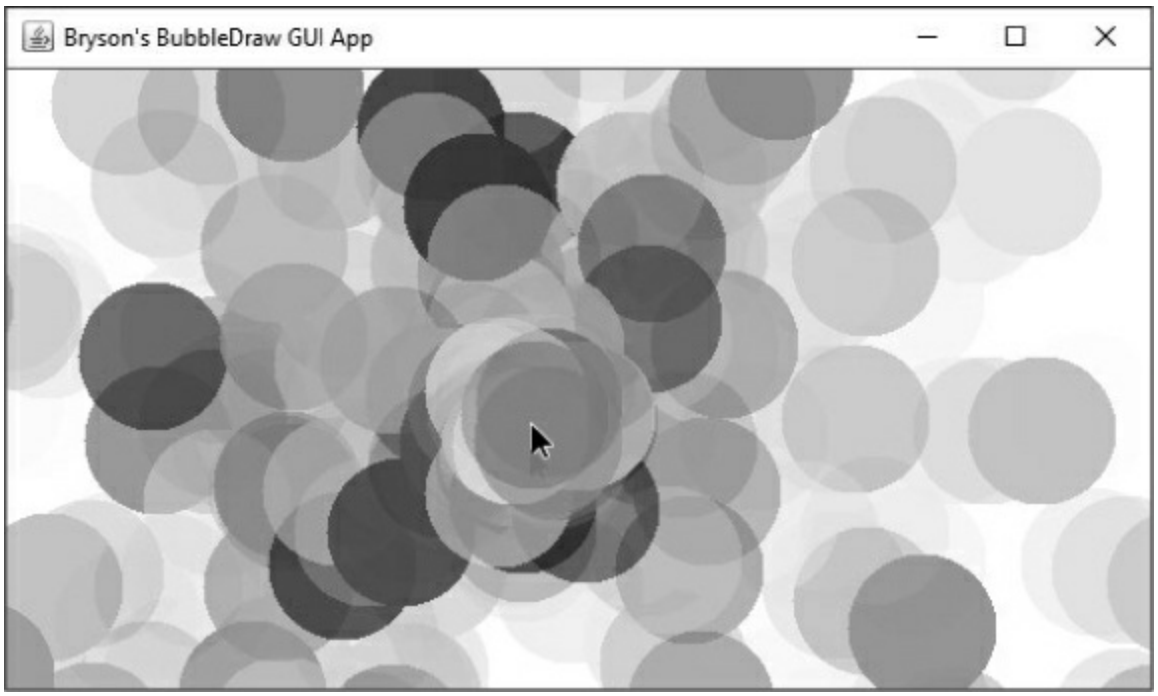


Figure 10-4: Changing both the x- and y-locations of each bubble makes the bubbles look like they're blowing in random directions from the mouse pointer as we drag.

The bubble's horizontal speed is how many pixels to the left or right it moves each frame. This is what determines the bubble's new x-location. Similarly, the bubble's vertical speed determines its new y-location. By just moving the bubble in the horizontal and vertical directions, we can make it move in any direction. Figure 10-5 shows how the horizontal speed and vertical speed combine to create the illusion that the bubble is moving diagonally.

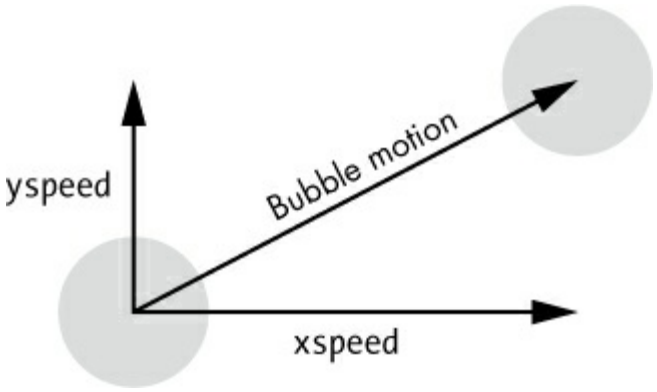


Figure 10-5: Changing a bubble's location in both the x- and y-directions quickly will make the bubble appear to move diagonally on the screen.

First, let's add variables to store how many pixels each bubble should travel in the x- and y-directions each time the screen is redrawn. Add these two lines to the top of the `Bubble` class:

```
private class Bubble {
   --snip--
   private Color color;
❶ private int xspeed, yspeed;
❷ private final int MAX_SPEED = 5;
```

At ❶, we've declared two integer variables: `xspeed` for the number of pixels the bubble will move horizontally each time the screen updates and `yspeed` for the number of pixels the bubble will move vertically. At ❷, we add a *constant* called `MAX_SPEED` for the maximum number of pixels a bubble should move at one time. Constants are named values similar to variables, but constants don't change inside a program, so we declare them as `final` to tell Java that the value of the constant is permanent. We also name constants in all uppercase as a convention so we can tell them apart from regular variables.

We'll give each bubble a random `x` and `y` speed using the `rand.nextInt()` method, just as we did for the bubble's color. Add these two lines to the `Bubble()` constructor:

```
public Bubble(int newX, int newY, int newSize) {
    x = newX;
    y = newY;
    size = newSize;
    color = new Color(rand.nextInt(256),
            rand.nextInt(256),
            rand.nextInt(256),
            rand.nextInt(256) );
    xspeed = rand.nextInt(MAX_SPEED * 2 + 1) - MAX_SPEED;
```

```
        yspeed = rand.nextInt(MAX_SPEED * 2 + 1) - MAX_SPEED;
    }
```

We need the `x` and `y` speed values to allow the bubble to move in any direction, but we have only two variables for four directions (left or right, up or down). We can handle this by using both negative and positive values. When the `xspeed` is negative, the bubble will move toward the left, and when it's positive, the bubble will move to the right. The `yspeed` will make the bubble move upward when negative and downward when positive. To make the `xspeed` and `yspeed` ranges span negative and positive values, I've multiplied `MAX_SPEED` by `2` and added `1`, which is equal to `11` since `5 * 2 + 1 = 11`. This makes `rand.nextInt(MAX_SPEED * 2 + 1)` equivalent to `rand.nextInt(11)`, which will return a number between `0` and `10`. By subtracting `MAX_SPEED` from this value, you'll get a result between `-5` and `+5` since `0 - 5 = -5`, and `10 - 5 = 5`.

Finally, we need to change the `update()` function to move the bubble to its new location each time the screen is redrawn. Replace the statement `y -= 5;` with the following two statements:

```
    public void update() {
        x += xspeed;
        y += yspeed;
    }
```

Instead of moving up five pixels every time the screen is redrawn, each bubble will move horizontally by the `xspeed` and vertically by the `yspeed` values we randomly generated for that bubble. The result is a colorful explosion of bubbles everywhere we drag the mouse!

Save and run the program with these changes, and you'll see an animated effect like the one we saw in Figure 10-4. The effect of moving the bubbles in the x- and y-directions over time gives each bubble the illusion of both a random speed and direction.

One curious behavior that you might notice is that some bubbles, such as the ones in the center of Figure 10-6, stay put. This is because we're randomly generating numbers between `-5` and `+5` for the `x` and `y` speed, and sometimes a bubble's `xspeed` and `yspeed` will both be `0`. When this happens, the bubble won't move at all.

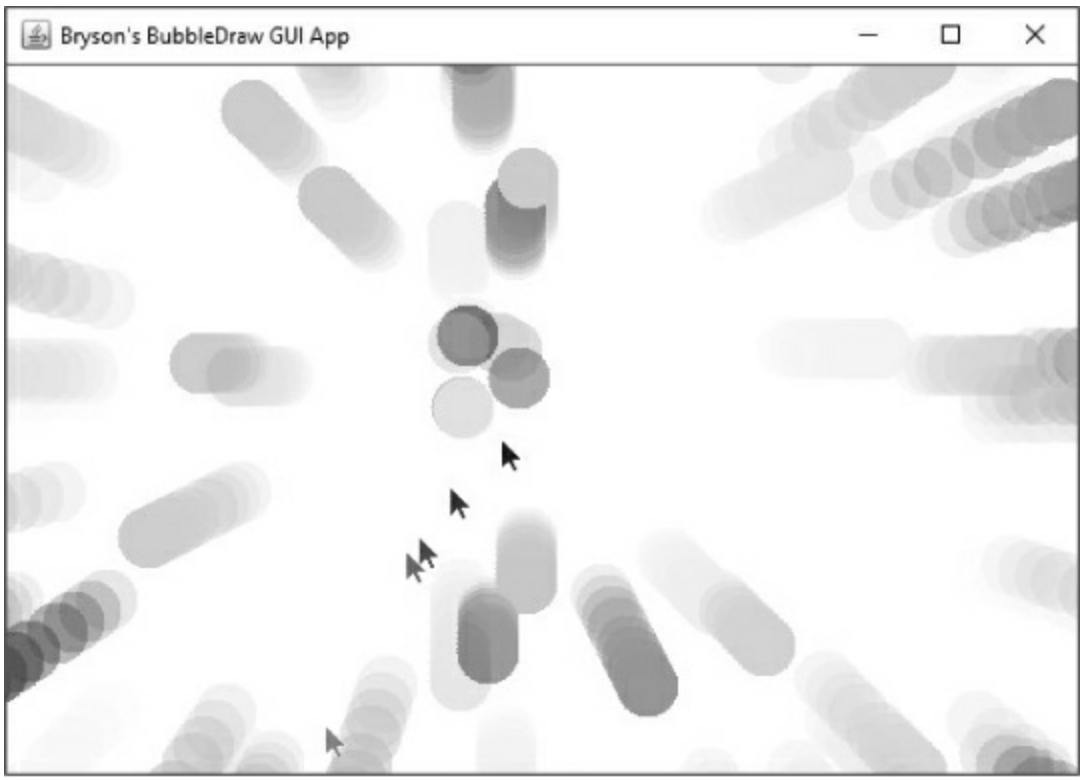


Figure 10-6: Because the random speed values can be 0, bubbles like the ones near the center here will stay put.

You can avoid having stuck bubbles by checking whether both `xspeed` and `yspeed` are equal to `0` and changing one or both of them—this is Programming Challenge #1 on page 244.

Now that our bubbles are floating away as we draw them, there are a couple of functions we might want to add to the app: the ability to pause the animation and to clear the screen. It's time to build a GUI interface within our animated, graphical app.

Building a GUI for Our Animated Drawing App

The BubbleDrawGUI app is graphical, but it doesn't have an interface like our other GUI apps. A Pause/Start button and a Clear button, shown in Figure 10-7, would make it easy for the user to understand and interact with the app, so let's add those next.

Figure 10-7: The Pause/Start and Clear buttons

Setting Up the GUI Panel and Buttons

In the Package Explorer pane, right-click *BubblePanel.java* and select **Open With ▸ WindowBuilder Editor**. Click the Design tab, and you should see the GUI design view.

First, let's add a `JPanel` to serve as the container for the Pause/Start and Clear buttons, as well as any other GUI components you might want to add later. In the Palette pane, under Containers, select **JPanel**. Then, hover your mouse over the design preview to the right and click the `BubblePanel` design preview to place a new `JPanel` onto the black background.

Alternatively, you can add a new `JPanel` by clicking the **javax.swing.JPanel** entry under the Structure pane and inside the Components pane to the left. You'll see a very small gray `JPanel` appear at the top of the black `BubblePanel` design preview.

Next, let's place the Pause/Start and Clear buttons. In the Palette pane, scroll down to the Components section, select the **JButton** component, and then hover and click inside the small `JPanel` we just added to place the first `JButton`. Enter **`Pause`** as the button's text, either directly in the GUI preview or in the Properties pane. (You'll see why we refer to it as the *Pause/Start* button shortly.)

Follow the same steps for the clear button: select **JButton** in the Palette pane, click inside the `JPanel` to place the button, and enter **`Clear`** for the button's text.

If it's difficult to place the buttons in the `JPanel` because it is too small, select **JButton** in the Palette and then click the panel in the Structure pane to the left to place each button inside the panel, as shown expanded in Figure 10-8. Name the buttons `btnPause` and `btnClear` either as you place them or by changing the `Variable` property in the Properties pane.

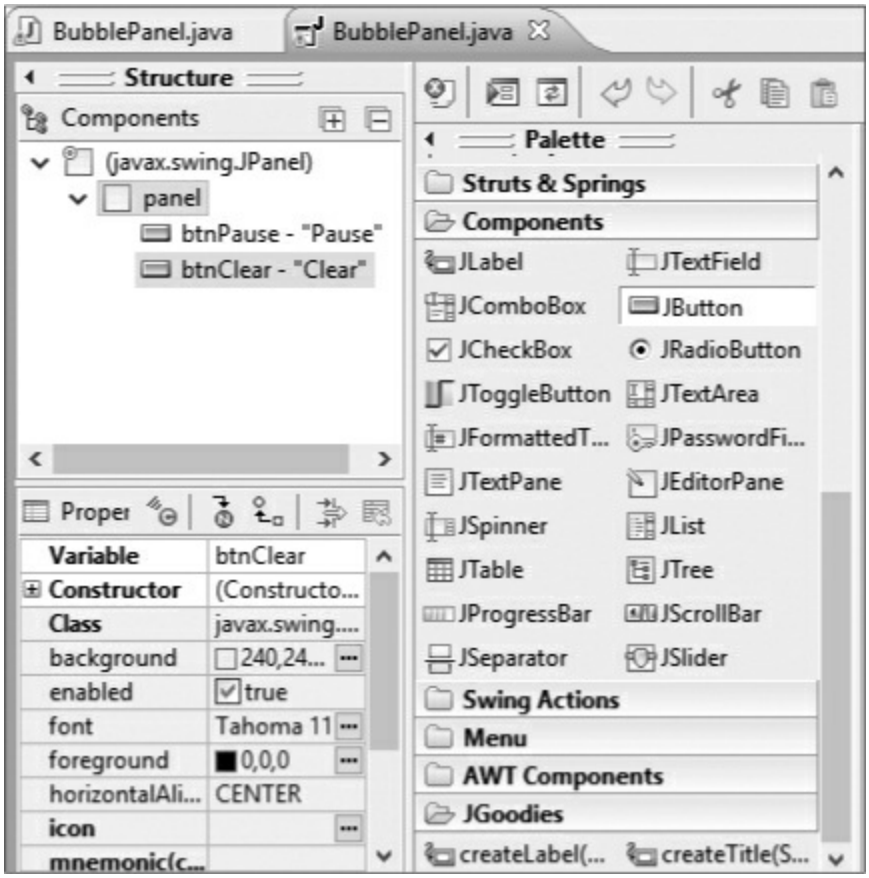


Figure 10-8: When building a GUI, you can select components from the Palette and add them directly to the Structure pane. Here, we've added btnPause and btnClear to the panel we just created.

The Structure pane is a useful way to add components to the GUI in situations like this, when we can't see the `JPanel` well in the design preview, or when we want to change the ordering or grouping of the components in the GUI. Figure 10-9 shows both buttons in the finished GUI.

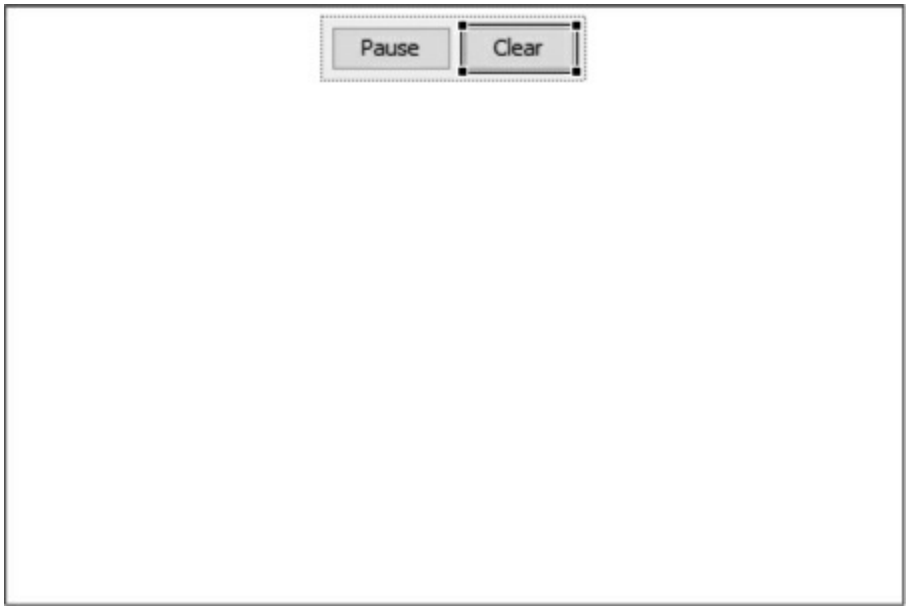


Figure 10-9: The Pause/Start and Clear buttons in the finished GUI

Now that we've added the Pause/Start and Clear buttons to the top of the drawing screen, it's time to code the buttons' event handlers.

Coding the Clear and Pause/Start Buttons

Let's start with the Clear button. One way to clear all the bubbles from the screen would be to reset the `bubbleList` variable to a new, empty list—that way, there are no bubbles to be drawn, and the user can begin painting afresh. To implement this behavior, double-click the **Clear** button (remember that double-clicking a button in the Design tab will create an event listener for the button and switch you back to the Source tab) and then add the following two lines inside the braces for `bubbleList`'s action listener:

```
JButton btnClear = new JButton("Clear");
btnClear.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
     ❶ bubbleList = new ArrayList<Bubble>();
     ❷ repaint();
    }
});
panel.add(btnClear);
```

At ❶, we clear the `bubbleList` variable by setting it equal to a new `ArrayList` of `Bubble` objects. This new list will be empty, so all we have to do is repaint the screen and we'll have a clean, black background just like when we opened the app. At ❷, we call the `repaint()` function to draw the new, empty screen.

Save and run the app. Then draw a few bubbles and click the **Clear** button to clear the screen.

Switch back to the Design tab and double-click the **Pause/Start** button to create another event listener. When the user clicks the Pause/Start button, we want to not only stop the animation by stopping the timer but also change the text on the button to *Start*. Then, when the user clicks again, we want to restart the timer to resume the animation and set the button's text back to *Pause*.

Enter the following code into the `actionPerformed()` method Eclipse provided when you double-clicked the Pause/Start button.

NOTE

Make sure your ActionEvent variable in the actionPerformed() method is named e, as highlighted in bold in this code.

```
JButton btnPause = new JButton("Pause");
btnPause.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     ❶ JButton btn = (JButton)e.getSource();
     ❷ if (btn.getText().equals("Pause")) {
         ❸ timer.stop();
         ❹ btn.setText("Start");
       }
       else {
         ❺ timer.start();
         ❻ btn.setText("Pause");
       }
    }
});
panel.add(btnPause);
```

At ❶, going from right to left, we use the `e.getSource()` method to find out which button was clicked, cast it to a `JButton` variable type, and store a link to the button as `btn`. The `getSource()` method is useful for finding out whether a GUI element has been clicked or changed, especially when you're writing event handlers for multiple elements at once. In this example, we can use `getSource()` to access the button's properties, such as its `text` property.

At ❷, we check whether the text on the button is equal to the string `"Pause"`. If so, we stop the timer ❸ to pause the animation, and then we change the text on the button to the string `"Start"` ❹.

If the text on the button wasn't `"Pause"`—in other words, if the animation was already paused and the text on the button had been changed to `"Start"` from a previous click—the event handler would go to the `else` statement instead and start the timer ❺ to resume the animation. Finally, the text on the Pause/Start button will change back to `"Pause"` ❻.

Save the file and run it once more. Now you can pause the animation, draw something, and then restart the animation to achieve the jaw-dropping, exploding-bubbles effect shown in Figure 10-10.

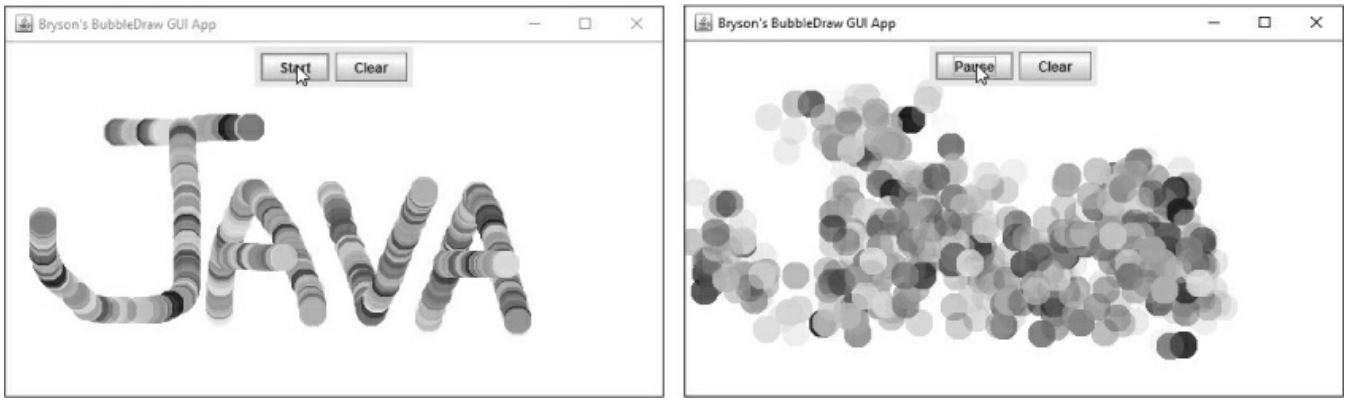


Figure 10-10: Pause the animation, draw a shape, and then start the animation again to see your drawing explode into colorful bubbles.

The BubbleDrawGUI app is a visually stunning animated app, and the buttons give the user more control over the drawing screen. But once the animation runs, bubbles float off the edges of the screen, never to return. What if we could make the bubbles bounce around inside the window so that they stayed around a bit longer?

Bouncing off the Walls with Collision Detection

Animation isn't just for flipbook cartoons and screensavers. There's another place you encounter animation all the time: computer games. Whether it's a mobile app or the latest online or console game, animation is how game developers give the user a sense of movement and action.

One useful game programming concept that we can add to the final version of the BubbleDrawGUI app is *collision detection*, which allows us to check whether two objects overlap, or *collide*, on the screen. You might use collision detection to tell the program what to do when a player shoots an enemy spaceship or kicks a football in a video game. In this app, we want to find out whether a bubble has reached the edge of the drawing screen so we can change the bubble's direction, making it appear to bounce off the edge of the screen back toward the center like the bubble in Figure 10-11.

Figure 10-11: The bubble bounces off the right edge of the window, courtesy of collision detection.

Collision detection can give virtual objects, such as our bubbles in BubbleDrawGUI, a more realistic appearance. In your favorite computer game, collision detection is what keeps your player from falling through the floor or walking through walls. These objects are imaginary—they're all just computer graphics, so they can't *actually* bump into one another—but collision detection creates the illusion that they are solid. So, if we give our bubbles a gentle bounce off the edges of the screen, they'll feel more like real objects to us. Let's see how that works.

A Soft Bounce

First, we'll break down how collision detection works for bubbles bouncing off the window's edges. We already know that each bubble has a pair of x -and y-coordinates for its location and an `xspeed` and `yspeed` for the number of pixels to move horizontally and vertically from one screen refresh to the next.

To figure out whether a bubble has collided with the window's edge, we need to know the x- and y-coordinates of the edges of the screen so we can compare them to the bubble's x- and y-coordinates. The left edge of the screen is the lowest `x` value, or `x==0`. And the top of the screen is the smallest `y` value, or `y==0`. But how about the right edge and bottom of the screen?

Every GUI component in Java inherits a pair of methods to return the width

and height of the component: `getWidth()` and `getHeight()`. Our drawing screen for the BubbleDrawGUI app is the `JPanel BubblePanel`. So, if we call the `getWidth()` and `getHeight()` functions inside `BubblePanel`, the maximum `x` value is `getWidth()`, and the maximum `y` value is `getHeight()`.

We check whether a bubble's `x` and `y` value has collided with an edge of the screen in the `Bubble` class's `update()` method. You might remember that the `update()` method is also where we change each bubble's x- and y-coordinates using `xspeed` and `yspeed` to give it the illusion of motion.

Let's be more precise about what we mean by a "bounce." Back in Figure 10-11, the bubble is moving toward the right edge of the screen, where `x==getWidth()`, meaning the bubble's `x` value is at the maximum x-coordinate, the width of the screen in pixels. To make the bubble look like it's bouncing, we change the direction it's moving by reversing the `xspeed`. The bubble was moving by some positive number of pixels each time it updated; after it touches the edge of the screen, we can make it move in the opposite direction by changing the sign of `xspeed`. In other words, we can make `xspeed` *negative* to make the bubble move to the left and away from the right edge of the screen after it bounces.

We can reverse the bubble's horizontal speed by setting `xspeed = -xspeed`, which makes `xspeed` the opposite of itself. So, an `xspeed` of `3` pixels per frame would change to an `xspeed` of `-3` pixels per frame after the bubble collides with the right edge of the screen, reversing its direction after the bounce.

We can do the same with the left edge of the screen, where `x==0`. If the bubble's `x` value allows the bubble to touch the left edge, setting `xspeed = -xspeed` flips the horizontal motion again: an `xspeed` of `-3` would become `-(-3)`, or `+3`. This will make the bubble move to the right again, away from the left edge of the screen.

In *BubblePanel.java*, scroll all the way down to the bottom of the file, where we defined the `Bubble` class. Find the `update()` method and add the following collision detection code for the left and right edges of the screen:

```
public void update() {
    x += xspeed;
    y += yspeed;
    if (x <= 0 || x >= getWidth())
        xspeed = -xspeed;
}
```

If the bubble's `x` value goes below `0`, or if `x` goes past the width of the screen,

the bubble must have touched either the left or right edge of the screen, so we change the `xspeed` to send the bubble bouncing in the opposite direction.

For the top and bottom edges, we'll do the same thing, but this time we change the `yspeed`:

```
    public void update() {
        x += xspeed;
        y += yspeed;
        if (x <= 0 || x >= getWidth())
            xspeed = -xspeed;
        if (y <= 0 || y >= getHeight())
            yspeed = -yspeed;
    }
```

If the bubble's `y` value goes below `0`, or past the height of the screen in pixels, we change `yspeed` to `-yspeed` to make the bubble move in the opposite direction.

Once you've made these additions to the `update()` method, save and run the file again. This time, you'll see the bubbles you create rebound softly off all four edges. You may notice a small issue, however: the bubbles move halfway off the screen in any direction before they seem to bounce, as shown in Figure 10-12.

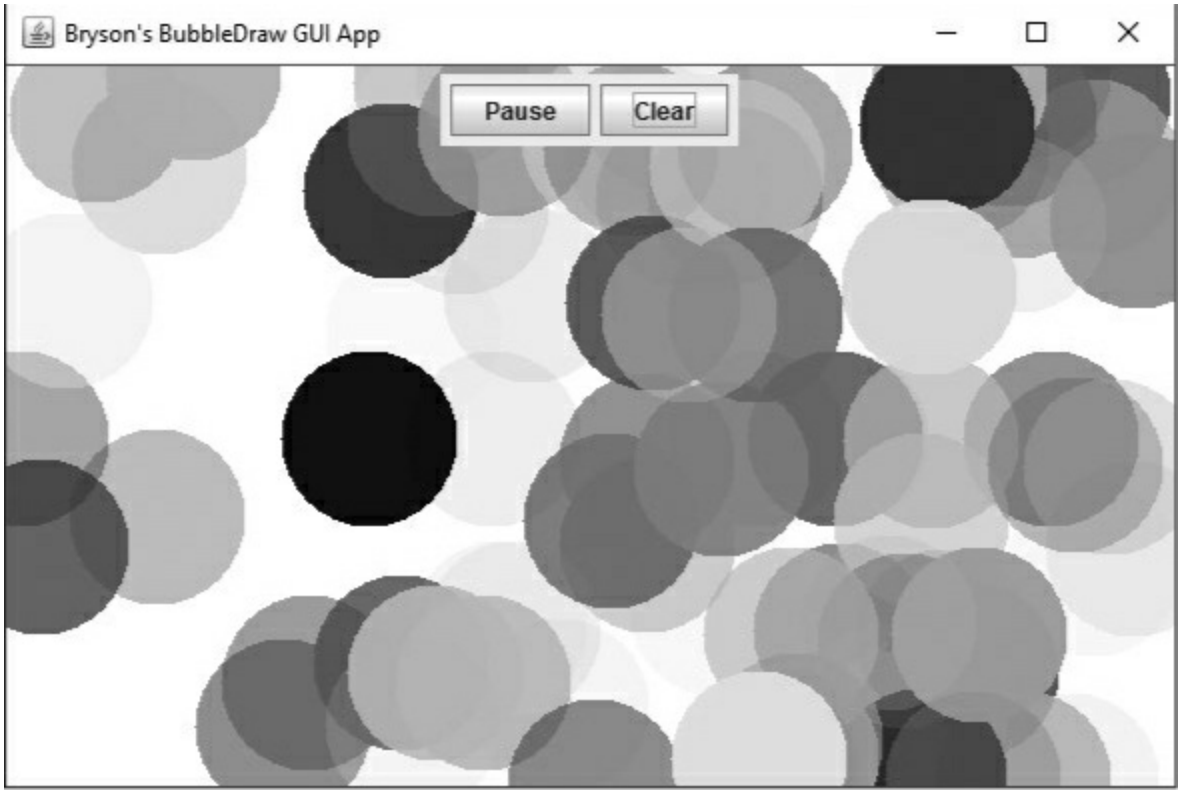


Figure 10-12: Around the edge of the window there are many bubbles that are almost halfway off the screen before they rebound.

This "soft" bounce happens because we're checking for a collision between the *center* of each bubble and the edges of the screen. Recall that we centered each bubble around the (x, y) coordinates of the user's click in Chapter 9, so each bubble's `x` and `y` values represent the location of the center of that bubble. To make the bubbles stay on the screen completely, we'll need to test for collisions between the outside edge of each bubble and the edges of the drawing window.

A Hard Bounce

To check for collisions with the edge of each bubble, we'll have to account for the distance from the center of each bubble to the edge, which is the bubble's radius (since each bubble is a perfect circle). The radius is the same as half of the `size` value of each bubble, or `size/2`. Modify the two `if` statements in the `update()` method to account for the size of each bubble as shown here:

```
    public void update() {
        x += xspeed;
        y += yspeed;
      ❶ if (x - size/2 <= 0 || x + size/2 >= getWidth())
            xspeed = -xspeed;
      ❷ if (y - size/2 <= 0 || y + size/2 >= getHeight())
            yspeed = -yspeed;
    }
```

At ❶, we subtract `size/2` from `x` to see whether the left edge of the bubble has touched the left side of the screen, which will be true if `x - size/2` is less than or equal to `0`. Division is performed before subtraction. Therefore `size/2` will be evaluated first and then subtracted from `x`, so we don't need to add parentheses around `size/2`. We also add `size/2` to `x` to see whether the right edge of the bubble is touching the right edge of the screen, which would mean `x + size/2` is greater than or equal to `getWidth()`. At ❷, we make the same changes to check for the top edge (`y - size/2`) and bottom edge (`y + size/2`) of each bubble to see whether they are touching the top or bottom, respectively, of the drawing window.

Save your program and run it again. Now all the bubbles you create, big and small, bounce solidly off the edges of the window, as shown in Figure 10-13.

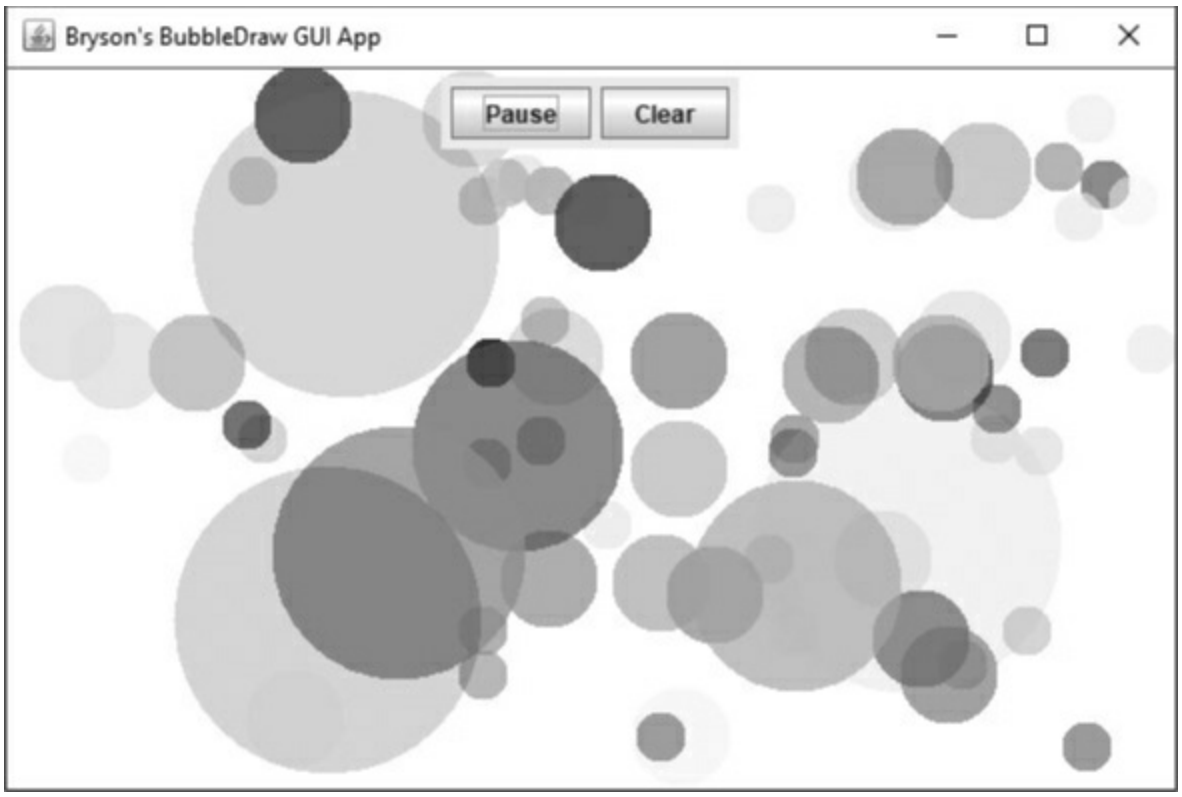


Figure 10-13: Now our bubbles make a "hard" bounce against the window's edges, for a more solid appearance.

Click the maximize button on the app's title bar or double-click the title bar to expand the window. You'll see that the bubbles expand to bounce off the edges of the drawing window even when the app is fullscreen. We used the `getWidth()` and `getHeight()` methods to determine the right and bottom edges, and those methods will always return the current width and height, so feel free to resize the app as you draw.

Now let's make one final addition to give the user even more control via the GUI.

Adding a Slider to Control the Animation Speed

At this point, we've given the user control over pausing and clearing the screen, as well as the ability to create big and small bubbles. Let's also give them control over the animation speed by providing a slider that changes the timer's delay, as shown in Figure 10-14.

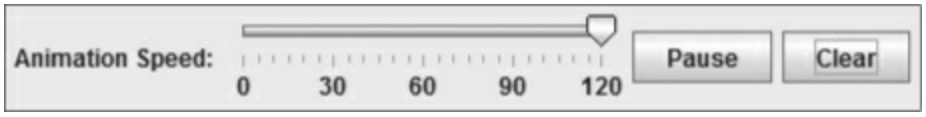


Figure 10-14: Adding a slider will allow the user to speed up or slow down the animation.

First, switch back to the Design tab and add a `JLabel` and `JSlider` to the GUI control panel. Scroll down under the Palette to find the Components section and select **JLabel**. Click inside the small panel just before the Pause/Start button in your GUI design preview to place the label. Change the label's text to `Animation Speed:`.

Next, click the **JSlider** component on the Palette. Then click between the Animation Speed label and the Pause/Start button to place the slider, as shown in Figure 10-15.

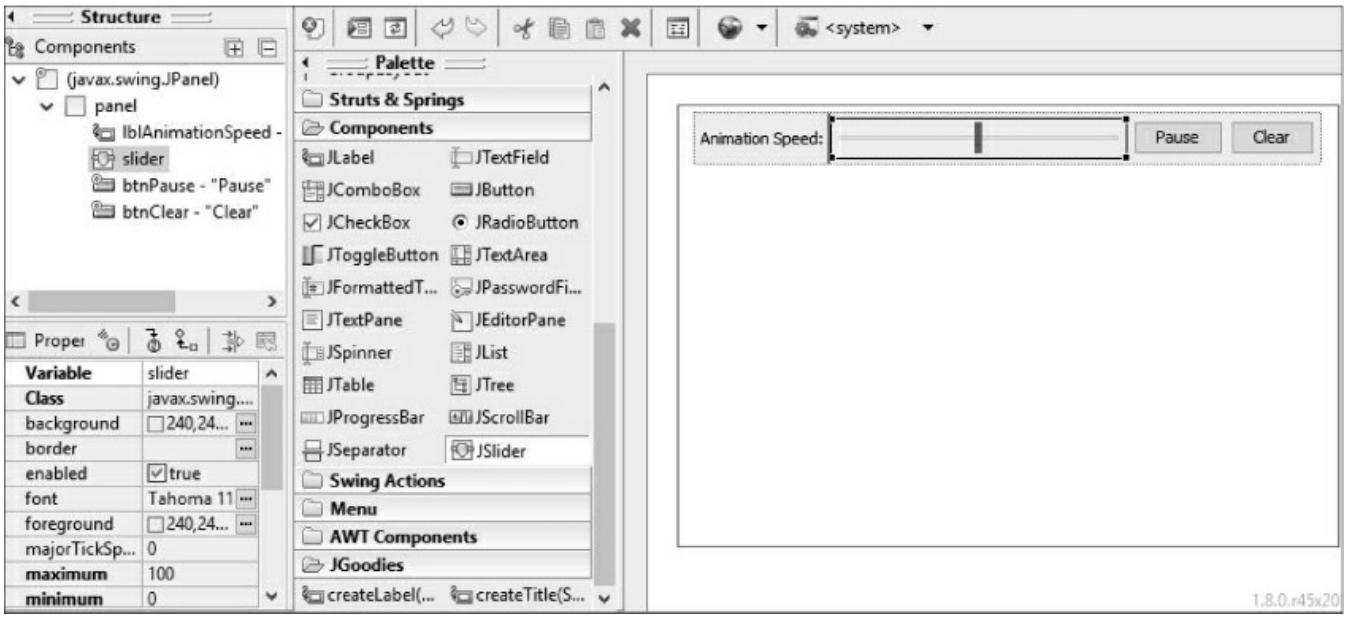


Figure 10-15: Add a label and slider to the app's GUI control panel in the design view.

You may notice that the `JPanel` for all the GUI components grows as we add elements. If you click the panel in the Structure pane, you'll see that the `Layout` property has the default value `java.awt.FlowLayout`. This layout expands to fit as many GUI elements as you place inside it. We used the `AbsoluteLayout` for the Hi-Lo guessing game and Secret Messages apps because we wanted to place elements in specific positions. In this drawing app, we can be more flexible, and the `FlowLayout` is perfect for adding GUI components on the fly.

Customizing the Slider

We'll change a few properties to customize the slider next, so let's figure out what we want the slider to look like. The slider should allow the user to change the animation speed easily and intuitively. In other words, if the user slides the animation speed to 0, the animation should slow down almost to a stop. If the user slides the animation speed all the way to the right, the animation should go really fast.

Monitors usually refresh the screen between 30 and 120 times per second, with 60 Hz (short for *hertz*, a measure of frequency per second) being the most common refresh rate. If we animate the bubbles faster than 120 times per second, your monitor likely won't be able to show all of the individual frames in the animation. So it makes sense to set our maximum speed value on the slider to 120 frames per second.

The number of frames per second, abbreviated *fps*, is often a measure of the smoothness of the animation. A game that runs at 60 fps on your computer will look smoother than a game that runs at 30 fps.

Select the slider in your design preview. In the Properties pane in the lower left, set the range by specifying a **maximum** of `120` and a **minimum** of `0` (the default). To prepare the labels and tick marks, set **majorTickSpacing** to `30`, set **minorTickSpacing** to `5`, and select the checkboxes next to **paintLabels**, **paintTicks**, and **paintTrack** to make all three values `true`. Finally, change the **value** property to `30`, the default number of frames per second. Figure 10-16 shows the Properties pane with all of our customized values, along with a preview of the `JSlider` after the changes.

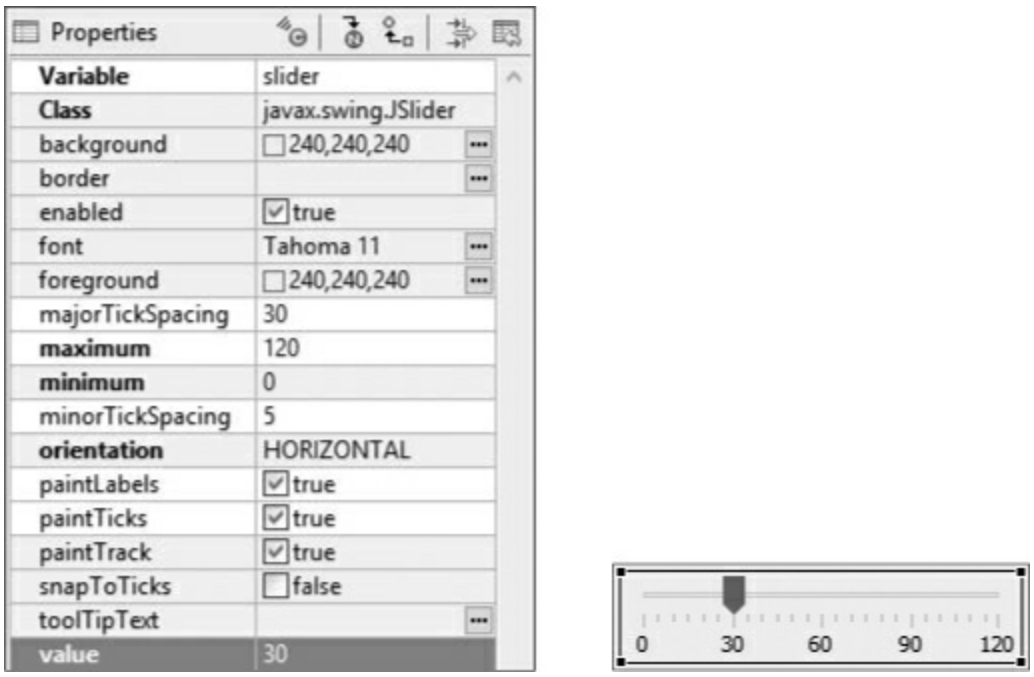


Figure 10-16: The Properties pane for the Animation Speed slider with customized values (left); a preview of the slider showing the customizations (right)

The slider is labeled from 0 to 120 in increments of 30 because you checked `paintLabels` and set `majorTickSpacing` to `30`. The small tick marks show up between those values because you set `minorTickSpacing` to `5` and checked `paintTicks`. We've got a customized slider ready to change the animation speed, so now let's edit the code to put the slider to work.

Implementing the Slider Event Handler

In the Design tab's design preview, right-click the slider and select **Add event handler ▸ change ▸ stateChanged**. Eclipse will add a `ChangeListener` with a `stateChanged()` method like the one we used in the Secret Messages app's slider implementation. The code looks like the following:

```
JSlider slider = new JSlider();
slider.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
    }
});
```

First, we need to declare the `JSlider` at the top of the class so that we'll be able to access the slider's value inside the `stateChanged()` event handler's code. Scroll up to the top of the `BubblePanel` class and add the following line just below the `timer` and `delay` variables:

```
public class BubblePanel extends JPanel {
    Random rand = new Random();
    ArrayList<Bubble> bubbleList;
    int size = 25;
    Timer timer;
    int delay = 33;
    JSlider slider;
```

Next, scroll back down to the slider code and remove the `JSlider` type declaration from the beginning of the first line:

```
        slider = new JSlider();  // Remove "JSlider" from beginning of line
        slider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
            }
        });
```

When the user changes the slider's position, we want to change the speed of the animation by changing the length of delay between each timer event. To do this, we'll need to get the speed value from the slider, convert the speed into a number of milliseconds, and then set the timer's delay to the new value. Add the following three lines inside the braces for the `stateChanged()` method:

```
      slider = new JSlider();
      slider.addChangeListener(new ChangeListener() {
          public void stateChanged(ChangeEvent arg0) {
              ❶ int speed = slider.getValue() + 1;
              ❷ int delay = 1000 / speed;
              ❸ timer.setDelay(delay);
          }
      });
```

At ❶, we use the `getValue()` method to get the speed value from the slider and store it in an integer variable named `speed`. Notice, though, that we add 1 to the slider value here. We're doing this to prevent a division-by-zero error at ❷, where we divide `1000` by the `speed` to determine the number of milliseconds of delay between each frame. The slider can go all the way down to 0, but by adding 1 to its value, we prevent the delay from causing an error: `1000/0` throws a division-by-zero exception, but `1000/1` gives us a very slow 1,000 milliseconds between frames that feels like the animation is nearly stopped. This means that when the user moves the slider to 0, the animation won't actually stop. In order to stop the animation completely, they'll need to click the Pause/Start button instead.

Finally, we update the timer by setting the new delay value in milliseconds ❸. This will slow down or speed up the animation by changing the amount of time between timer events.

Save the file and run it. Move the slider back and forth, and you'll see you now have control over the speed of the animation.

The BubbleDrawGUI app is our most interactive, entertaining, visually engaging app so far—a graphical, animated drawing app with a GUI interface that gives the user complete control over the animation. Play around with it for a while—and congratulate yourself on a job well done!

SHARE AND SHARE ALIKE

This would also be a great app to share with friends. To export a runnable JAR file from Eclipse, go to **File ▸ Export**. Then, expand the *Java* folder and click **Runnable JAR file**. Click **Next** and, under **Launch configuration**, click the drop-down list and choose **BubbleDrawGUI – BubbleDrawGUI**.

Under **Export destination**, click **Browse**, and then select the destination folder you'd like to save your finished app in, perhaps *Desktop*. Give your program file a name, such as *Bryson's BubbleDraw.jar*. Click **Save**, then **Finish**.

Find where you saved your JAR file, run it, and share it with a friend. Even if your friend doesn't have Eclipse, as long as they have Java installed, they'll be able to run your BubbleDrawGUI app. Enjoy!

What You Learned

We built on the BubbleDraw app from Chapter 9 to produce an animated GUI version with bouncing bubbles. Here are some of the skills we've added in this chapter:

- Combining a graphical app with a GUI interface
- Copying a project and pasting a new version in the Package Explorer in Eclipse

- Renaming a class or object by refactoring
- Using transparency by setting the alpha component in RGBA colors
- Creating, setting, and starting a `Timer` object
- Handling `Timer` object events
- Creating an animation by moving graphical objects with a timer
- Making virtual objects bounce via collision detection
- Using `getWidth()` and `getHeight()` to find the edges of the window
- Using a slider to change a timer's delay
- Varying an animation's speed by changing the `delay` property of a timer

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned, as well as to expand your programming skills. Visit the book's website at *https://www.nostarch.com/learnjava/* for sample solutions.

#1: No Bubble Left Behind

One problem we noted in the chapter was that some bubbles get a random speed of `0`, causing them to seem stuck in place while the other bubbles float away. This happens when the bubble's `xspeed` and `yspeed` are both set to `0`. For this challenge, add some code to make sure that no bubbles get a random speed of `0`. To do so, you'll test the `xspeed` and `yspeed` values to see whether they're both equal to `0`. If they are, all you need to do is set the two values to something else, like `1`.

HINT

Add the if statement inside the Bubble() constructor, after the xspeed and yspeed values are created.

```
private class Bubble {
    --snip--
        public Bubble(int newX, int newY, int newSize) {
```

```
--snip--
    xspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
    yspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
    if  // Add your code here
}
```

Make your changes, save and run the file, and voilà—no more stranded bubbles!

#2: Flexi-Draw!

Because of the random speed values, the bubbles move in every direction, which is the effect we originally wanted. But what if we wanted to draw shapes and have them stay together?

Setting the speed of every bubble to the same fixed value creates a funky, flexible, twisty effect as the objects bounce off the edges of the screen, as shown in Figure 10-17.

Figure 10-17: Giving all bubbles the same xspeed and yspeed creates a flexible, bouncy effect that keeps shapes together as they rebound and twist off the edges of the screen.

For this challenge, create a new copy of the app so that the original version is still available. Copy and paste the BubbleDrawGUI project in the Package Explorer pane, renaming it FlexiDraw or another app name of your choice. In the *BubblePanel* file, change the `Bubble()` constructor so that instead of creating random `xspeed` and `yspeed` variables, it sets them equal to the same value, perhaps like this:

```
        xspeed = yspeed = 2;
```

This line of code makes use of an interesting feature of the assignment operator (`=`). This is called a *chained assignment*, because both `xspeed` and `yspeed` are being assigned the value of `2`, and the equal sign allows us to assign the same value to several variables in a chain sequence.

You can pick a higher or lower number for your speed value. The important point to note is that we've replaced the random speeds with a fixed starting speed and direction for every bubble. Each bubble will move right two pixels and down two pixels the first time it's drawn, so the bubbles will form groups as you draw.

Save the file, right-click the *FlexiDraw* folder in the Package Explorer, and select **Run As ▸ Java Application**. Pause the animation to draw, and then press **Start** to see your shapes bend, twist, and bounce as they move around the screen! You can even draw as the animation runs for a cool spiral effect.

#3: PixelDraw 2.0

For this challenge, you'll reuse the PixelDraw code from Chapter 9 (Programming Challenge #2 on page 219). Adding the pixelated effect to the animated drawing program will allow you to draw square pixel shapes and animate them. Combine this effect with the FlexiDraw trick from the previous challenge exercise, and you've got a bouncing, bending *Minecraft*-esque drawing app like the one shown in Figure 10-18.

Figure 10-18: The PixelDraw 2.0 effect (left); the same shape after bouncing off the sides and twisting upside down and backward (right)

Pause the animation to keep the blocks in a clean, gridlike shape like the example in Figure 10-18. Draw with the animation going to achieve a stacked, 3D look like the example in Figure 10-19.

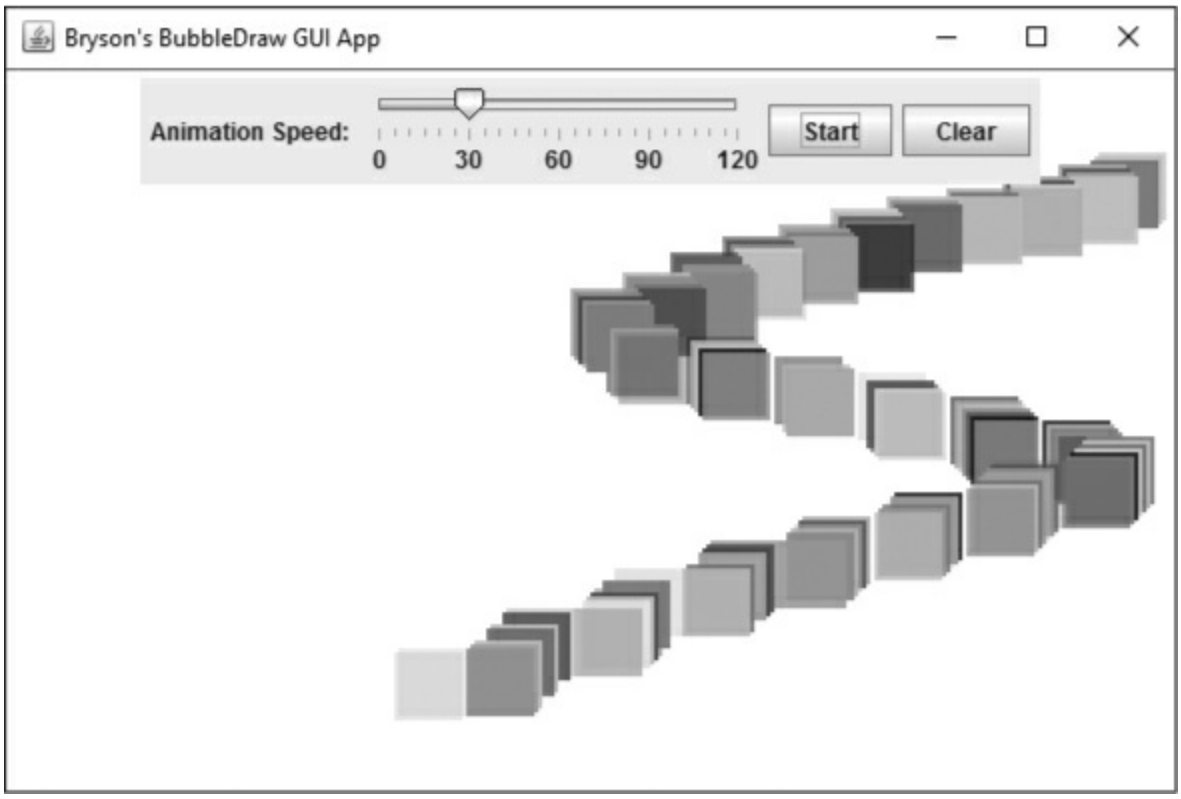


Figure 10-19: The PixelDraw 2.0 code creates a stacked, 3D effect if you drag the mouse while the animation is running.

Here's a hint to help with the math: to achieve the blocky style, you need to "divide" the screen into a grid based on the size of each block, and then draw a block at that grid location. This time, try changing your `x` and `y` variables at the top of the `Bubble()` constructor to something like the following:

```
x = (newX / newSize) * newSize + newSize/2;
y = (newY / newSize) * newSize + newSize/2;
```

The first part of the formula, `(newX / newSize) * newSize`, divides the screen into grid blocks based on `newSize`, the size of the current bubble. In order to align the x- and y-positions with a `newSize` by `newSize` grid, we need them to be multiples of `newSize`. For the x-coordinate, we do this by dividing `newX` by `newSize`, which results in an integer without decimal points that we multiply by `newSize` to make an integer that is a multiple of `newSize`. This places the x-coordinate of the bubble at the edge of the grid block the user clicks in. For example, if `newSize` is `10`, the division and multiplication "snap" the x-coordinate to `10` by `10` gridlines by making `x` a multiple of `10`. If we stopped there, this would result in a bubble with its center at the point where the gridlines meet. Since we want the bubble to be contained within a grid block instead, the second part (`+ newSize/2`) shifts

the bubble to the center of that block.

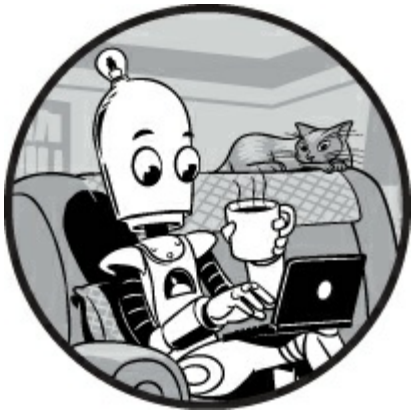
Finally, draw blocks instead of bubbles by changing the `draw()` method to fill rectangles instead of ovals using `fillRect()`.

That's it! But you can—and should!—make additional changes to customize the new app even further and make it your own. Change the `JFrame` title in the *BubbleDrawGUI.java* file or even refactor/rename the files if you'd like. The sky's the limit!

After making those changes, you'll be able to draw beautiful, blocky, pixelized, and animated creations. Take a screenshot and tweet it to your friends. Use the hashtag #JavaTheEasyWay or tag me @brysonpayne, and I'll retweet it to a few thousand of my friends as well!

11

MAKING BUBBLEDRAW A MULTITOUCH ANDROID APP



Our final app will be a multitouch Android version of BubbleDraw that enables the user to draw bubbles with the touch of a finger—or all 10 fingers!

The processors in Android devices are usually much smaller and slower than desktop CPUs. If you've ever gotten an "App Not Responding" error, you've seen what happens when an app consumes too much of a device's processing power. So, instead of a timer, this version of the app will use a new approach to animation called *threading*, which will make the app use less processing power. Threading is what makes it possible for us to run multiple apps at the same time, or *multitask*.

The new BubbleDraw app will also make use of *multitouch*. Take a look at Figure 11-1, which shows bubbles pouring out of several different locations, where my younger son Max touched his fingers to the screen.

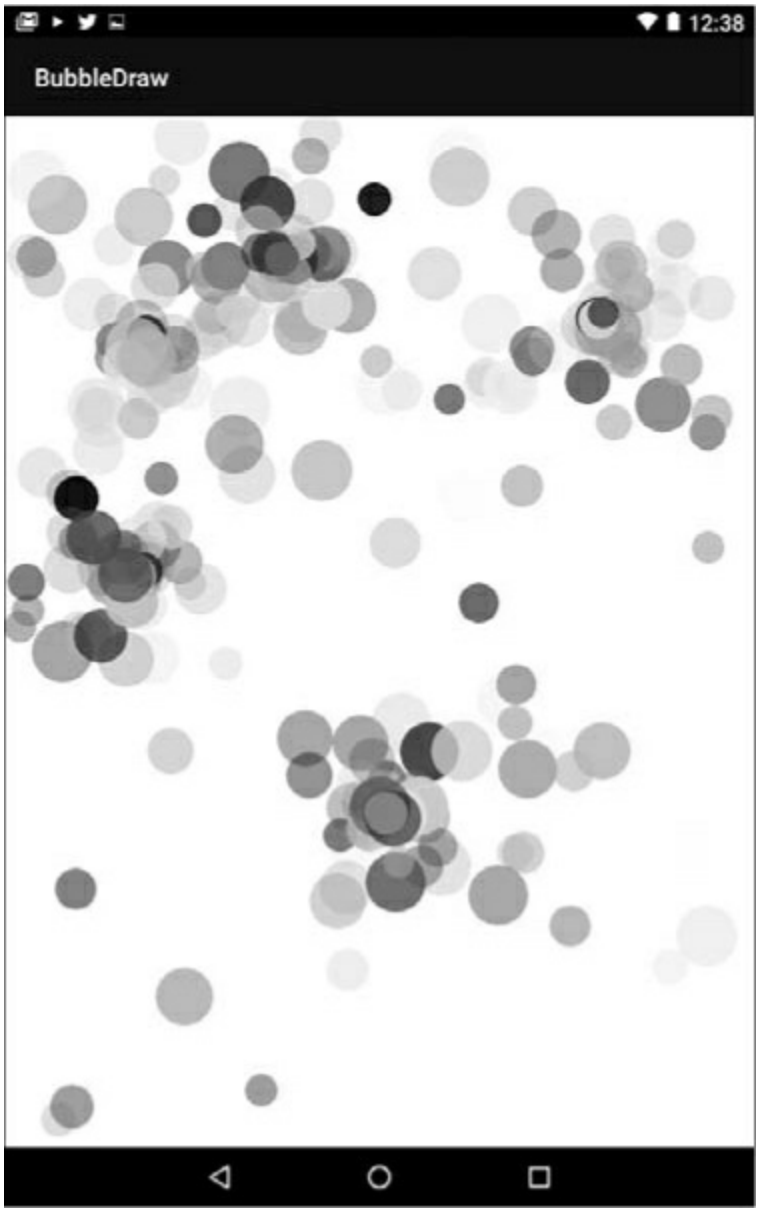


Figure 11-1: The Android version of the BubbleDraw app will use multitouch so that the user can draw bubbles on multiple locations of the screen at once.

The mobile BubbleDraw app will reuse a lot of the features from the desktop and GUI versions, such as the source code from the `Bubble` class we finished building in Chapter 10. There are a couple of differences in how we draw graphics on Android, however, and since we're also adding threading and multitouch, you'll need to learn some new app-building techniques. You'll develop these new skills using the BubbleDraw app as a foundation.

Setting Up the BubbleDraw Project

Open Android Studio, close any open projects, and click **Start a new Android Studio project**. In the Create New Project window, enter **BubbleDraw** in the Application Name field, leave Company Domain the same (*example.com* or your website name), and then click **Next**.

This time, we want to choose a newer API level than we used for our previous apps. Both the Hi-Lo guessing game and Secret Messages apps used GUI interfaces that would work on older Android devices. In this app, however, we'll need the `drawOval()` method to draw the bubbles, which requires API level 21 or higher. Select **API 21: Android 5.0 (Lollipop)** as the Minimum SDK.

An additional difference is that instead of the basic activity we used in the Hi-Lo guessing game and Secret Messages apps, in this app we'll use an empty activity because we don't need a basic GUI interface. Instead of a regular GUI app layout, we're going to create an interactive, touch-enabled drawing canvas. On the Add an Activity to Mobile screen, choose **Empty Activity** and click **Next**.

We'll keep `MainActivity` as the Activity Name, as we did in our previous apps, but uncheck the **Backwards Compatibility** checkbox and click **Finish**. Turning off backward compatibility will keep our code simpler.

As in the previous two desktop versions of the BubbleDraw app, we're going to use two Java files to keep our bubble code separate from the main app code. Once your project opens, click the Project tab along the left edge of the screen to display the Project Explorer pane, if it's not already visible. Then, select the **Android** tab at the top of the Project Explorer pane. Under *app* ▸ *java*, find the *main* BubbleDraw package (don't select the *androidTest* or *test* packages). Right-click your BubbleDraw package in the Project Explorer and select **New ▸ Java Class**, as shown in Figure 11-2.

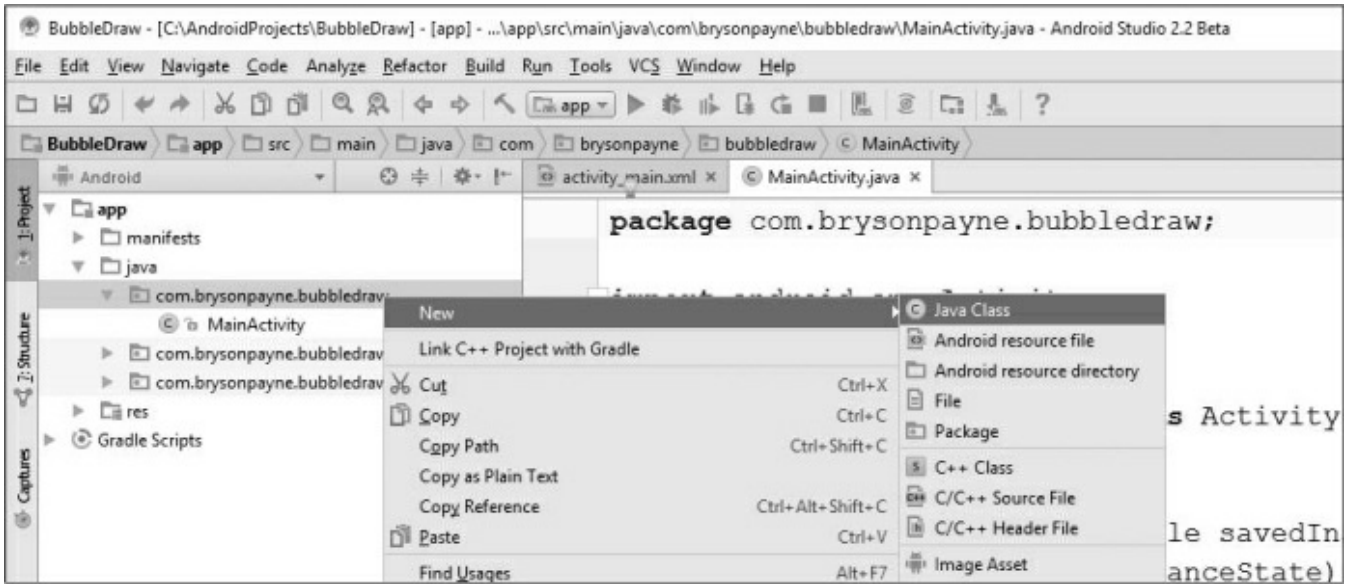


Figure 11-2: Add a new Java class to your main BubbleDraw package for the bubble code.

In the Create New Class pop-up window, name the new class **`BubbleView`**. In Android, a `View` is any GUI component. The `BubbleView` class serves the same function as the `BubblePanel` class did in the desktop app. All our bubble drawing code will go in this class.

Android Studio's Create New Class window allows us to set the superclass and interfaces easily. First, we'll make our new `BubbleView` class inherit the ability to draw graphics easily. In the **Superclass** text field, begin typing **`ImageView`** and then click **ImageView (android.widget)** in the autocomplete drop-down list to make the new class's parent class `ImageView`. This will display as `android.widget.ImageView` after you click it.

Next we'll implement an `OnTouchListener` interface to enable our app to handle touch events, similar to the mouse events we used in the previous versions of the app. In the **Interface(s)** text field, begin typing **`OnTouchListener`** and then click **OnTouchListener (android.view.View)** in the autocomplete drop-down list. Once you click it, it will display as `android.view.View.OnTouchListener`.

Click **OK**. You should now see the `BubbleView` class inside your *com.<yourdomain>.bubbledraw* package. Double-click the `BubbleView` class in the Project Explorer to begin editing the file. The `BubbleView` class may be underlined in red to let you know it's missing some code, but we'll fill in those required pieces as we code the app in the next few sections.

Creating the BubbleView Constructor

Double-click the *BubbleView.java* tab to expand it to fullscreen in the app for easier editing. Let's begin building the `BubbleView` class by adding variables similar to those in the `BubblePanel` class. Just as in our desktop and GUI versions of the app, we'll need a random number generator and an `ArrayList` for the bubbles the user has drawn, as well as some integer variables for the default bubble size and the animation delay in milliseconds.

Adding the Animation Variables

We'll want random colors and speeds for our bubbles, so inside the opening brace for the `BubbleView` class, begin typing **`private Random`** and then click **Random (java.util)** in the autocomplete drop-down list.

We'll click the automatic code completion list items every time we add a new type of object in our code. Remember, Android Studio's code completion feature not only helps you code faster, it reduces errors from mistyped or misspelled class names and `import` statements.

Finish declaring your **`private Random rand = new Random();`**. Then, add each of the variables shown in the following code. Check your `import` statements afterward to make sure they match the ones here:

```
package com.yourdomain.bubbledraw;  // Note: your package name may differ
import android.widget.ImageView;
import android.view.View;
import java.util.ArrayList;
import java.util.Random;
public class BubbleView extends ImageView implements View.OnTouchListener {
 ❶ private Random rand = new Random();
 ❷ private ArrayList<Bubble> bubbleList;
 ❸ private int size = 50;
 ❹ private int delay = 33;
}
```

These four lines are similar to the variables we declared at the top of the `BubblePanel` class in Chapters 9 and 10, with a few changes for the Android version of the app. The declaration for a random number generator at ❶ is identical to our old version, because both use `java.util.Random`. The same goes for the line at ❷, where we declare an `ArrayList` of `Bubble` objects called

`bubbleList`. This will again be where we store the bubbles created by the user. The `Bubble` type specifier should show up in red, letting us know we haven't defined a `Bubble` class yet.

At ❸, we're declaring an integer variable for the default bubble size, but we've made it bigger for the Android app because of the smaller pixel size on mobile devices. Your mobile phone or tablet often has a much smaller screen size and denser resolution than your desktop computer, so we'll set our default bubble size to `50` to make the bubbles easier to see in the app. You can edit this line later to make your bubbles bigger or smaller depending on how you want them to look on your device.

At ❹, we're keeping the animation speed at 30 fps by setting the delay between frames to `33` milliseconds. Remember, to get the animation speed, we divide 1,000 milliseconds by the number of frames per second, 30 fps, to get the number of milliseconds per frame, 1,000 ÷ 30 = 33.

Both the graphics and animation will be slightly different on Android than they were on desktops, so we'll need to add two new variables:

```
public class BubbleView extends ImageView implements View.OnTouchListener {
    private Random rand = new Random();
    private ArrayList<Bubble> bubbleList;
    private int size = 50;
    private int delay = 33;
  ❶ private Paint myPaint = new Paint();
  ❷ private Handler h = new Handler();
}
```

The line at ❶ declares an `android.graphics.Paint` object called `myPaint`. Think of this as a paintbrush for drawing bubbles on the Android screen. You must have a `Paint` object to be able to draw shapes on an Android `Canvas`. Press ENTER after typing `Paint` to accept the code completion suggestion, or click `Paint` after typing it and press ALT-ENTER (or OPTION-ENTER) to automatically import the `android.graphics.Paint` class.

The line at ❷ declares another new type of variable, an `android.os .Handler` named `h`. Make sure you import the `android.os` version of the `Handler` class since there are other classes with similar names. This `Handler` object will enable us to work with threading to accomplish the animation, and you can think of it as analogous to the `Timer` in the desktop app. Unlike a `Timer`, however, the `Handler` will allow us to communicate with a *thread*, which is an individual process in a

multitasking environment where we might be running several apps at once. A `Handler` won't keep the CPU busy counting off the time between events as a `Timer` would; instead, it will release the CPU and allow other tasks to run until it's time to redraw another frame of animation.

Next, let's add a constructor to the app and start drawing bubbles.

Creating the BubbleView() Constructor

The next step will be writing the constructor for the `BubbleView` class. Below the variables we just declared, enter the constructor code as follows:

```
public class BubbleView extends ImageView implements View.OnTouchListener {
    private Random rand = new Random();
    private ArrayList<Bubble> bubbleList;
    private int size = 50;
    private int delay = 33;
    private Paint myPaint = new Paint();
    private Handler h = new Handler();
    public BubbleView(Context context, AttributeSet attributeSet) {
        super(context, attributeSet);
    }
}
```

Use automatic code completion to import `android.content.Context` and `android.util.AttributeSet`. Android uses these two classes to store information about the current application, and we need to import them to be able to call the `super()` method. The `super()` method sets up the app and drawing screen by calling the constructor of the parent class, `ImageView`.

For now, the only statement we'll add to the constructor is a line to initialize the `bubbleList`:

```
public BubbleView(Context context, AttributeSet attributeSet) {
    super(context, attributeSet);
    bubbleList = new ArrayList<Bubble>();
}
```

Initializing the `bubbleList` by setting it equal to a new, empty `ArrayList` of `Bubble` objects works the same way here as it did in the previous versions of the app. We'll be able to store new bubbles in `bubbleList` as the user touches the screen.

Preparing the Layout to Use BubbleView

Now that we've begun building the `BubbleView` class, it's time to tell our GUI layout file to display `BubbleView` when the app runs. In the Project Explorer, open *app* ▸ *res* ▸ *layout* ▸ *activity_main.xml* and switch to the Text tab at the bottom of the window.

Replace the contents of the *activity_main.xml* file with the following, substituting your app's package name in place of mine:

```
  <?xml version="1.0" encoding="utf-8"?>
❶ <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
    ❷ android:background="#000000"
    ❸ tools:context="com.yourdomain.bubbledraw.BubbleView">
    ❹ <com.yourdomain.bubbledraw.BubbleView
    ❺ android:layout_width="match_parent"
    ❻ android:layout_height="match_parent"
      />
</RelativeLayout>
```

This app uses the default `RelativeLayout` ❶, which will allow us to easily place other GUI components later. Much of the `RelativeLayout` properties are the same as the default, but we've added a `background` color of `#000000` ❷, or black.

When you replace the package name `com.`*yourdomain*`.bubbledraw` at ❸ and ❹ with your package name, Android helps you by making a code completion suggestion at ❹.

The line at ❹ places the `BubbleView` into the layout, and the lines at ❺ and ❻ tell the program to match the width and height of the *activity_main .xml* window, which is the parent object of `BubbleView`.

The *activity_main.xml* file is the default GUI layout view that your app loads when it runs. Telling *activity_main.xml* to load `BubbleView` as the only item in the layout and match the layout's width and height results in the `BubbleView` app taking up the full screen. So, with these changes, the *activity_main.xml* file now knows to call on `BubbleView` to show our bubble drawing canvas.

Speaking of bubbles, let's reuse the `Bubble` class from the desktop version of this app.

Modifying the Bubble Class

Open the *BubbleDrawGUI* project from Chapter 10 in Eclipse to access the old `Bubble` class. Open the *BubblePanel.java* class and scroll to the bottom of the file where we defined the `Bubble` class. Copy the class's entire source code, from `private class Bubble` all the way down to the next-to-last closing brace. The final brace in the file is the closing brace for `BubblePanel`, so make sure you copy only the closing braces for `update()` and the `Bubble` class.

Now that you've copied the `Bubble` class, switch back to Android Studio and place the cursor directly after the closing brace for the `BubbleView()` constructor. Press ENTER to insert a blank line before the final closing brace at the bottom of the file.

Most of the code from the desktop version of `Bubble` will work, but we'll need to make a few changes to account for differences in the way Android draws graphics. Let's start near the top of the `Bubble` class. In Android graphics, color values are stored as integers instead of `Color` objects, so change `private Color color` to `private int color` as shown here:

```
private class Bubble {
    private int x;
    private int y;
    private int size;
    private int color;
    private int xspeed, yspeed;
    private final int MAX_SPEED = 5;
```

All the other variables remain the same.

We'll also need to change the color entry in the `Bubble()` constructor. Delete `new Color` and replace it with `Color.argb`, as shown here:

```
public Bubble(int newX, int newY, int newSize) {
    x = newX;
    y = newY;
    size = newSize;
    color = Color.argb(rand.nextInt(256),
            rand.nextInt(256),
            rand.nextInt(256),
            rand.nextInt(256) );
    xspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
    yspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
}
```

Be sure to remove the keyword `new`, since `Color.argb()` doesn't create a new

object. Instead, the `Color.argb()` method will convert four *ARGB* values (*alpha*, *red*, *green*, and *blue*) into a single color integer that can be used to change paint colors in Android.

This is the first time we've used the `Color` class in this app, so it will show up in red in the Android Studio text editor. You can either manually add **`import android.graphics.Color;`** to the `import` statements at the top of the file, or you can click the word **Color** and press ALT-ENTER (or OPTION-ENTER) to have Android Studio perform the import for you. Pressing ALT-ENTER is similar to accepting a code completion suggestion, except that you can use ALT-ENTER to import a class even *after* you've typed your code.

Next, we'll need to change the entire `draw()` method in the `Bubble` class. Replace the `draw()` method we copied with the following:

```
        xspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
        yspeed = rand.nextInt(MAX_SPEED * 2) - MAX_SPEED;
    }
❶ public void draw(Canvas canvas) {
    ❷ myPaint.setColor(color);
    ❸ canvas.drawOval(x - size/2, y - size/2,
              x + size/2, y + size/2, myPaint);
    }
    public void update() {
```

At ❶, the `draw()` method accepts a parameter of type `android.graphics .Canvas` instead of `java.awt.Graphics`. Be sure to import the `Canvas` class either as you type or by clicking it and pressing ALT-ENTER (or OPTION-ENTER) afterward.

At ❷, we set the color for the `myPaint` object to use this bubble's `color` value.

The line at ❸ differs from the desktop version in several places. First, the command to draw an oval on an Android `Canvas` is `drawOval()` instead of `fillOval()`. Second, we specify the bounding box that will hold the oval using its *left*, *top*, *right*, *bottom* values instead of *left*, *top*, *width*, *height*. The left and top remain the same, at `x - size/2` and `y - size/2`. (Remember, we subtract half the width and height of the bubble to center it on the `(x, y)` location where the user touched the screen.) The right side of the bubble's bounding box is `x + size/2`, and the bottom of the bubble is `y + size/2`, as shown in Figure 11-3. Instead of using the width and height to calculate the bottom-right corner of the bounding box for the oval, as we did in the desktop version, Android requires us to specify the x- and y-coordinates of the bottom-right corner, or `x + size/2, y + size/2`.

Finally, the `drawOval()` method requires a `Paint` object, so we pass it `myPaint`.

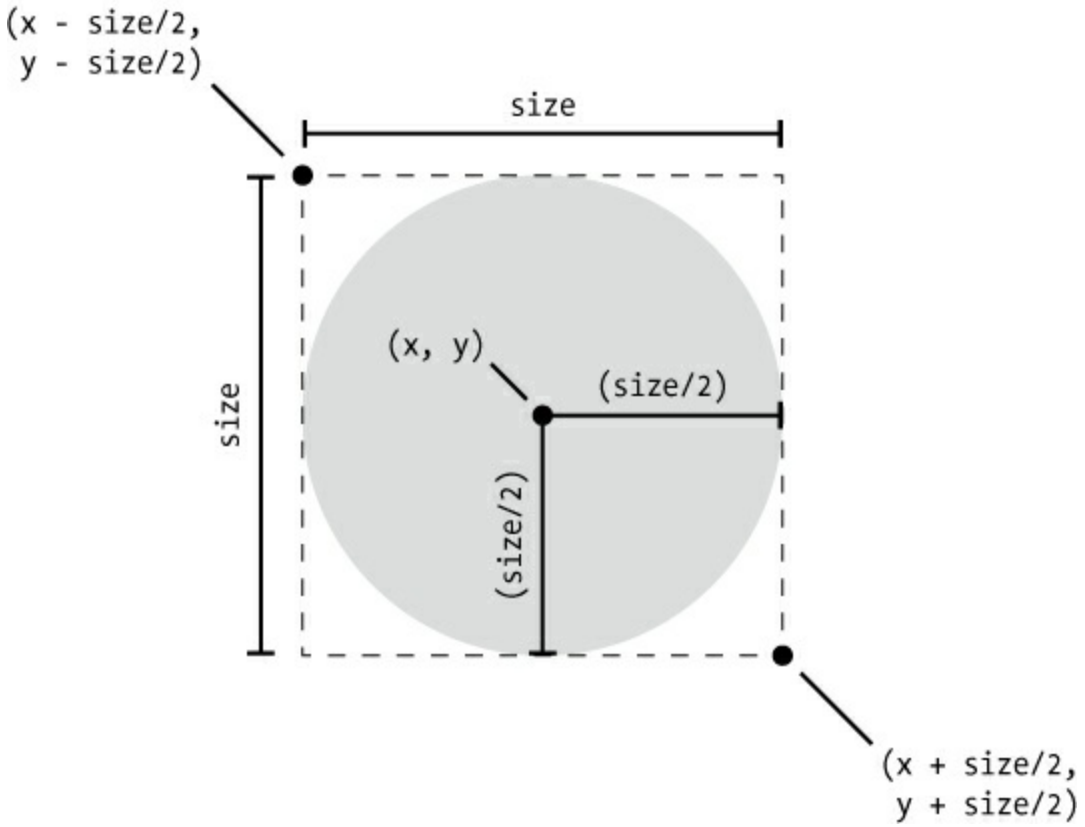


Figure 11-3: Android's drawOval() method takes the top-left and bottom-right coordinates of an imaginary bounding box instead of the top, left, width, and height values we saw in the desktop Swing toolkit.

Those are the only changes needed to port the `Bubble` class from desktop to Android. Save your file after making these changes. Next, we'll draw all the bubbles on the screen.

Drawing in Android with the onDraw() Method

We want to test the app's ability to draw bubbles on the screen, so next we'll add an `onDraw()` method to our `BubbleView` class. The `onDraw()` method in a `View` class is similar to the `paintComponent()` method in a `JPanel`: it tells Java what to draw whenever the screen is refreshed.

We want to draw the list of bubbles, so add the following code below the `BubbleView()` constructor in *BubbleView.java* and above the `Bubble` class:

```
public BubbleView(Context context, AttributeSet attributeSet) {
```

```
        super(context, attributeSet);
        bubbleList = new ArrayList<Bubble>();
    }
❶   protected void onDraw(Canvas canvas) {
      ❷ for (Bubble b : bubbleList)
            b.draw(canvas);
    }
    private class Bubble {
```

At ❶, the `onDraw()` method must be declared as protected with one parameter, a `Canvas` object, to work because it must exactly match the default `View.onDraw()` method that we're overriding. We need this method because it's required in all `View` subclasses and `BubbleView` is a child class of `ImageView`, which is a subclass of `View`. The `onDraw()` method will be called anytime the screen containing our `BubbleView` needs to be refreshed.

The inside of the `onDraw()` method reuses the `for-each` loop at ❷ that calls each bubble's `draw()` function. The line at ❷ could be read as follows: "For each `Bubble b` in `bubbleList`, draw `b` on the Android `Canvas`."

There are just a couple of steps left before we can test our app's ability to draw colorful bubbles. Let's knock those out to see an early beta test of the Android BubbleDraw app.

Testing BubbleDraw with 100 Bubbles

We wrote a short method called `testBubbles()` in the first version of the BubbleDraw desktop app. It looked like the following:

```
public void testBubbles() {
    for(int n = 0; n < 100; n++) {
        int x = rand.nextInt(600);
        int y = rand.nextInt(400);
        int size = rand.nextInt(50);
        bubbleList.add( new Bubble(x, y, size) );
    }
    repaint();
}
```

The purpose of `testBubbles()` was to see whether we could draw bubbles on the screen before we had implemented the mouse and timer event handlers. Let's do the same for the Android version of the app.

Adding testBubbles()

First, let's add a slightly modified version of the `testBubbles()` function just below the `onDraw()` method in *BubbleView.java*:

```
protected void onDraw(Canvas canvas) {
    for (Bubble b : bubbleList)
        b.draw(canvas);
}
public void testBubbles() {
    for(int n = 0; n < 100; n++) {
     ❶ int x = rand.nextInt(600);
     ❷ int y = rand.nextInt(600);
     ❸ int s = rand.nextInt(size) + size;
     ❹ bubbleList.add( new Bubble(x, y, s) );
    }
  ❺ invalidate();
}
private class Bubble {
```

The first two lines are identical to the Eclipse version of `testBubbles()`. These lines declare the function and set up a `for` loop to run 100 times. At ❶, we keep the range of `x` values the same and set it to `600`, but you can make this larger if you know the resolution of your device. At ❷, we change the range for the random `y` value to `600` for the bubble's vertical location.

At ❸, we generate larger bubbles by adding a random number between `0` and `size` to the default value `size`. In this case, we'll have bubbles ranging from 50 to 100 pixels in diameter.

At ❹, we create a new `Bubble` object using the three random values we just created and add it to the `bubbleList`.

Finally, at ❺ we're using a new function, `invalidate()`, which works similarly to the `repaint()` function in the desktop version of BubbleDraw. It tells Java that the screen needs to be updated, or refreshed. The `invalidate()` function clears the screen and calls the `onDraw()` method, which will draw all the bubbles in `bubbleList`.

Now that we've defined the `testBubbles()` method, we just need to call that method from the `BubbleView()` constructor to try it out:

```
public BubbleView(Context context, AttributeSet attributeSet) {
    super(context, attributeSet);
    bubbleList = new ArrayList<Bubble>();
```

```
    testBubbles();
}
```

Now, when the app loads, the `testBubbles()` method will be called to populate the `bubbleList` with 100 random bubbles.

Fixing the OnTouchListener Error

There's only one issue left to fix before we can test our app: the `BubbleView` class is still underlined in red to let us know there's a possible compile error. Mouse over the line `public class BubbleView`, and you'll see an error that tells us we're missing an `onTouch()` method for the `OnTouchListener`. In other words, this error reminds us that we've defined the `BubbleView` class as implementing an `OnTouchListener`, but we haven't yet added an `onTouch()` method to handle touch events.

To fix this error, mouse over it and click the red lightbulb warning icon. In the drop-down menu under the red lightbulb, click **Implement methods**. A pop-up window will appear, asking you to select which methods to implement, as shown in Figure 11-4.

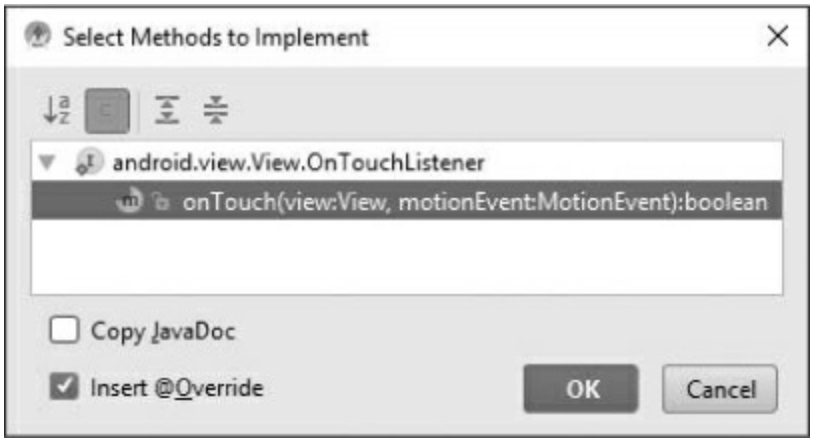


Figure 11-4: Android Studio implements the onTouch() method to complete the OnTouchListener.

Click **OK**, and Android Studio will insert an `onTouch()` method into your code to clear the error:

```
@Override
public boolean onTouch(View view, MotionEvent motionEvent) {
    return false;
}
```

If the code Android Studio inserted doesn't match what you see here, correct

the parameter names in your code to match `view` and `motionEvent` as shown. This is the method that we'll modify to handle touch events later, but for now let's give our app a test run to see it draw 100 bubbles!

Running the BubbleDraw App

Press the green run button to test your app. In the Select Deployment Target window, choose your emulator or device, as we did in "Running the App on the Android Emulator" on page 94. I'm selecting my **Nexus 6P** emulator, shown in Figure 11-5.

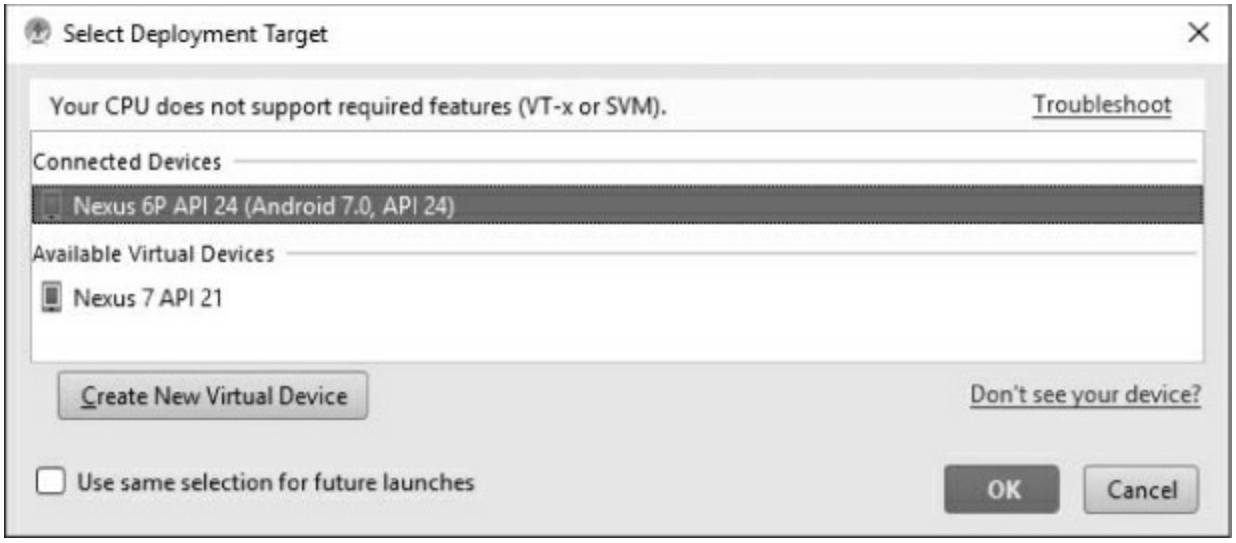


*Figure 11-5: Press the run button to compile and run your app. Then select your emulator or device and click **OK**.*

Click **OK** to start your device and deploy the BubbleDraw app we've developed to this point. You should see bubbles fill the upper-left portion of your screen, as shown in Figure 11-6. The bubbles appear in the upper left because we only generated x- and y-coordinates between 0 and 600 and most Android devices can display 1,000 or more pixels in each direction. Remember, in Java, the (0,0) coordinate is in the upper left.

Figure 11-6: A successful test run of the BubbleDraw app to this point; 100 bubbles appear in the upper left of the screen.

Similar to the first test run of the desktop version of the app, there's no animation or touch interaction, because we haven't added them yet. But our app runs on the emulator and draws our bubbles correctly on the screen.

Next, let's add animation to make the bubbles move. And then we'll add the final touch—touch!

Using Threaded Animation and Multitasking in Java

The threading we'll use to achieve smooth animation in the BubbleDraw Android app is just like the threading you'd add to any multitasking app in Java. We mentioned earlier that one advantage of using threading for animation is that it doesn't tie up the processor while it waits between redrawing each frame. But threading can also be used in any other apps you develop that run multiple processes at once, such as apps that query a database or upload a file in the background. Using threading allows your app to do background tasks without freezing up the GUI interface while it waits for a process to finish.

Threading is especially important on devices such as phones and tablets that have limited processing power. Unresponsive apps are especially annoying, and threading will help prevent the BubbleDraw app from becoming one.

The `Handler h` object that we created earlier in the chapter is what lets us communicate with a thread. In this app, we'll create a thread that executes the animation by updating the positions of all the bubbles. Then, we'll use our handler, `h`, to tell the thread when to run. This will allow the handler and thread to function like our `Timer` animation in the previous chapters, but without hogging the CPU between frames.

There are two ways to add threading to your applications in Java: by extending the `Thread` class or by implementing the `Runnable` interface. We'll choose the `Runnable` approach.

A class that implements `Runnable()` needs a `run()` method that tells the thread what to do when the thread is running. For BubbleDraw, we want the `run()` method to perform the animation by moving the bubbles and redrawing the screen.

Let's create a `Runnable` object called `r` and place it right below the `BubbleView()` constructor:

```
  public BubbleView(Context context, AttributeSet attributeSet) {
      super(context, attributeSet);
      bubbleList = new ArrayList<Bubble>();
      testBubbles();
  }
❶ private Runnable r = new Runnable() {
      @Override
```

```
    ❷ public void run() {
      }
❸ };
  protected void onDraw(Canvas canvas) {
```

As you type the second half of the declaration at ❶, a code completion suggestion will pop up to complete the new `Runnable()` object. Accept the `java.lang.Runnable` code completion, and Android Studio will automatically add the `public void run()` method stub for you ❷. Notice the semicolon after the closing brace for the `Runnable` object ❸—this is required because we're defining `r` and assigning it to a new `Runnable` object all at the same time. The semicolon at ❸ actually completes the statement we started back at ❶. The automatic code completion doesn't add the semicolon after the closing brace at ❸, though, so make sure you've added it to avoid a compiler error.

Now we need to add the code inside the `run()` method to tell Java what we want to happen every time our `Runnable` thread `r` is called:

```
private Runnable r = new Runnable() {
    @Override
    public void run() {
     ❶ for(Bubble b : bubbleList)
        ❷ b.update();
     ❸ invalidate();
    }
};
protected void onDraw(Canvas canvas) {
    for (Bubble b : bubbleList)
        b.draw(canvas);
}
```

At ❶, we use the `for-each` statement again to loop through every `Bubble b` in `bubbleList`. At each iteration of the loop, we call b.update() ❷ to update the location of each bubble for the next frame of animation.

At ❸, we call the `invalidate()` function after the loop has finished running to clear the screen and tell Java to redraw the view by calling the `onDraw()` method.

The final step to adding thread-based animation is to connect the `Handler h` to the `Runnable` thread `r`. We'll do this at the end of the `onDraw()` method:

```
protected void onDraw(Canvas canvas) {
    for (Bubble b : bubbleList)
        b.draw(canvas);
```

```
    h.postDelayed(r, delay);
}
```

The `postDelayed()` method sends a message from the handler to our thread `r`, telling it to run again after a delay of 33 milliseconds, the value of `delay`.

Save these changes and run the app again. You'll see the 100 test bubbles slowly spread out to fill the screen, as shown in Figure 11-7.

Even at a rate of 30 frames per second, your fastest bubbles probably seem quite slow. This is because of the larger number of pixels on your Android device or emulator. You may recall that we set a `MAX_SPEED` in the `Bubble` class of just 5 pixels per frame. The Nexus 6P phone that we're emulating has a screen resolution of 1,440 × 2,560, meaning it would take more than 500 frames, or more than 15 seconds, for the fastest bubbles to travel the full length of the screen.

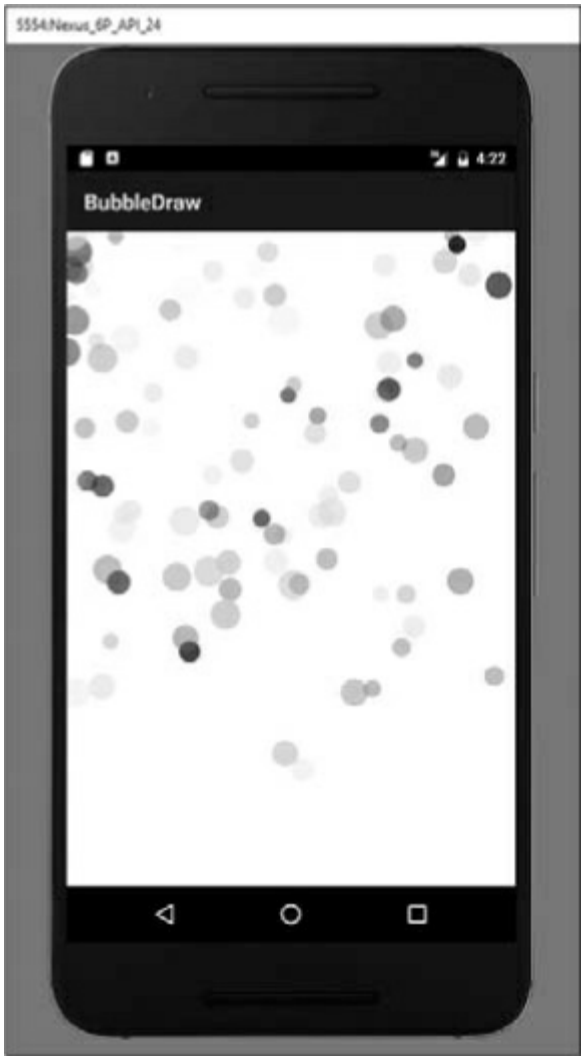


Figure 11-7: We have liftoff! Our bubbles are finally animated, courtesy of threaded animation.

Let's speed that up just a bit by changing `MAX_SPEED` to a higher value, like 15 pixels per frame:

```
private class Bubble {
    private int x;
    private int y;
    private int size;
    private int color;
    private int xspeed, yspeed;
    private final int MAX_SPEED = 15;
```

Save the file and run it again. Your bubbles will move around the screen more fluidly now. Feel free to increase or decrease the value to suit your taste.

Now that we've implemented the animation, it's time to add touch to make the app respond to the user.

Using Touch to Draw with Your Finger

One feature that made the desktop BubbleDraw app so much fun was the fact that we could click and drag the mouse to draw bubbles anywhere we wanted. We're going to bring that same level of enjoyment to the Android app, and then we'll turn it up to 11 when we add multitouch!

You've already seen where we need to add the touch event handler code—in the `onTouch()` function we added earlier.

To handle touch events, first we need to determine the location of the user's touch. Then we'll add a bubble at that location.

To find out the x- and y-coordinates of the user's touch, we can use `motionEvent.getX()` and `motionEvent.getY()`. Let's add the full `onTouch()` method and then break it down:

```
public boolean onTouch(View view, MotionEvent motionEvent) {
 ❶ int x = (int) motionEvent.getX();
 ❷ int y = (int) motionEvent.getY();
 ❸ int s = rand.nextInt(size) + size;
 ❹ bubbleList.add( new Bubble(x, y, s) );
 ❺ return true;
}
```

At ❶, we get the x-location of the user's touch with the method

`motionEvent.getX()`. Notice, however, that we have to cast the value to an integer —the `motionEvent.getX()` returns a floating-point value in Android, so we cast it with `(int)`. The y-location ❷ works the same way, and at ❸, we generate a random size as we did in `testBubbles()` and store it in `s`.

At ❹, we create a `Bubble` object with the given `x`, `y`, and `s` values and add it to `bubbleList`.

The last line, at ❺, requires a bit of explanation. Notice that the return type of the `onTouch()` method is a `boolean`. This means that the `onTouch()` method must return a `true` or `false` value. In Android, the `onTouch()` method should return `true` if you've fully handled the touch event. If you want Android to handle a touch event, such as scrolling or zooming, after your `onTouch()` event handler is finished, return `false`.

For the drawing app, we don't need Android to scroll the screen when the user swipes—we've handled the touch event completely for our app by adding a bubble where the user touched, so we return the value `true`.

The last step is similar to the steps for the mouse listener in the desktop versions of BubbleDraw: we need to add the listener in the constructor. Scroll up to the `BubbleView()` constructor, comment out the `testBubbles()` function, and add the line of code shown here:

```
public BubbleView(Context context, AttributeSet attributeSet) {
    super(context, attributeSet);
    bubbleList = new ArrayList<Bubble>();
    // testBubbles();
    setOnTouchListener(this);
}
```

We comment out the call to `testBubbles()` because we don't need the 100 test bubbles anymore. We're going to add bubbles by touching the screen on our Android device or by clicking and dragging the mouse to simulate a touch on the Android emulator. The `setOnTouchListener(this)` statement adds a listener for touch events, telling Java that `this`, an object of the `BubbleView` class, will handle the touch events.

With these changes, we're ready to try out our app. Save your code and run the app on your emulator. Click and drag on the emulator window to simulate dragging your finger across the screen. You'll see bubbles flow from the location you touch, as shown in Figure 11-8.

You can also run this on your physical Android device. We'll review how to

do that in the next section, after adding the ability to handle multiple touch events on the screen at the same time.

Figure 11-8: Click and drag the mouse to simulate a one-finger touch on the emulator, and a stream of bubbles will flow out.

Using Multitouch to Draw with 10 Fingers at a Time!

You may have played an app that makes use of multitouch, like a two-player game in which you control objects on one side of the screen with one or more fingers, while your friend controls objects on the other side of the screen. If you have, you understand the awesome power that multitouch brings to apps and games.

The great news is that the code to handle multiple touch events in Android is almost as simple as the code to handle a single touch! In fact, we'll just add a statement and modify a couple of lines in `onTouch()`, and our app will handle

multitouch events.

The method that tells us how many touch events are happening at the same time is `getPointerCount()`. It returns how many *pointers*—touch events or fingers—are on the screen in the current `MotionEvent`.

We can add a `for` loop to add bubbles for every pointer in the current touch event:

```
public boolean onTouch(View view, MotionEvent motionEvent) {
  ❶ for (int n = 0; n < motionEvent.getPointerCount(); n++) {
      ❷ int x = (int) motionEvent.getX(n);
      ❸ int y = (int) motionEvent.getY(n);
        int s = rand.nextInt(size) + size;
        bubbleList.add(new Bubble(x, y, s));
  ❹ }
   return true;
}
```

At ❶, we add a `for` loop that increments the variable `n` from `0` up to the number of pointers in the current touch event. The function `motionEvent.getPointerCount()` returns the number of pointers. If there is only one touch event, `getPointerCount()` will return `1`, and the loop will run only once, for `n = 0`. For two touch events, `n` will get the values `0` and `1`, and so on.

At ❷, we modify the `motionEvent.getX()` method by inserting an `n` in the parentheses after `getX()`. Pointers are numbered in touch events, so passing the variable `n` as an argument to `motionEvent.getX()` will get the x-location of the *n*th touch pointer in the current touch event. So, `getX(0)` will return the x-location of the first touch, `getX(1)` will return the x-coordinate of the second touch, and so on. At ❸, we do the same for the y-coordinates of each touch with `getY(n)`. Finally, don't forget to close the brace of the `for` loop at ❹.

That's all it takes! Java and Android make processing multitouch events easy.

Testing Multitouch Events on an Android Device

Save your code so that we can run the app again. This time, we're going to run the app on a physical Android device. Unfortunately, we can't easily simulate multitouch events with a single mouse on the Android emulator, so you'll need to run the app on a real phone or tablet.

First, plug your Android device into the computer running Android Studio

using a USB cable. Allow USB debugging from the computer on the device (see "Running the App on a Real Android Device" on page 100). Close the BubbleDraw app in your emulator, if it's running. Then, press the run button or go to **Run ▸ Run 'app'**.

In the Select Deployment Target window, find and select your device—mine is the Asus Nexus 7—and click **OK**. Android Studio will recompile the app and deploy it to your device.

When the app starts, it looks like a black screen with "BubbleDraw" in the title bar. Once you place one or more fingers on the screen, though, things get much more interesting, as shown in Figure 11-9.

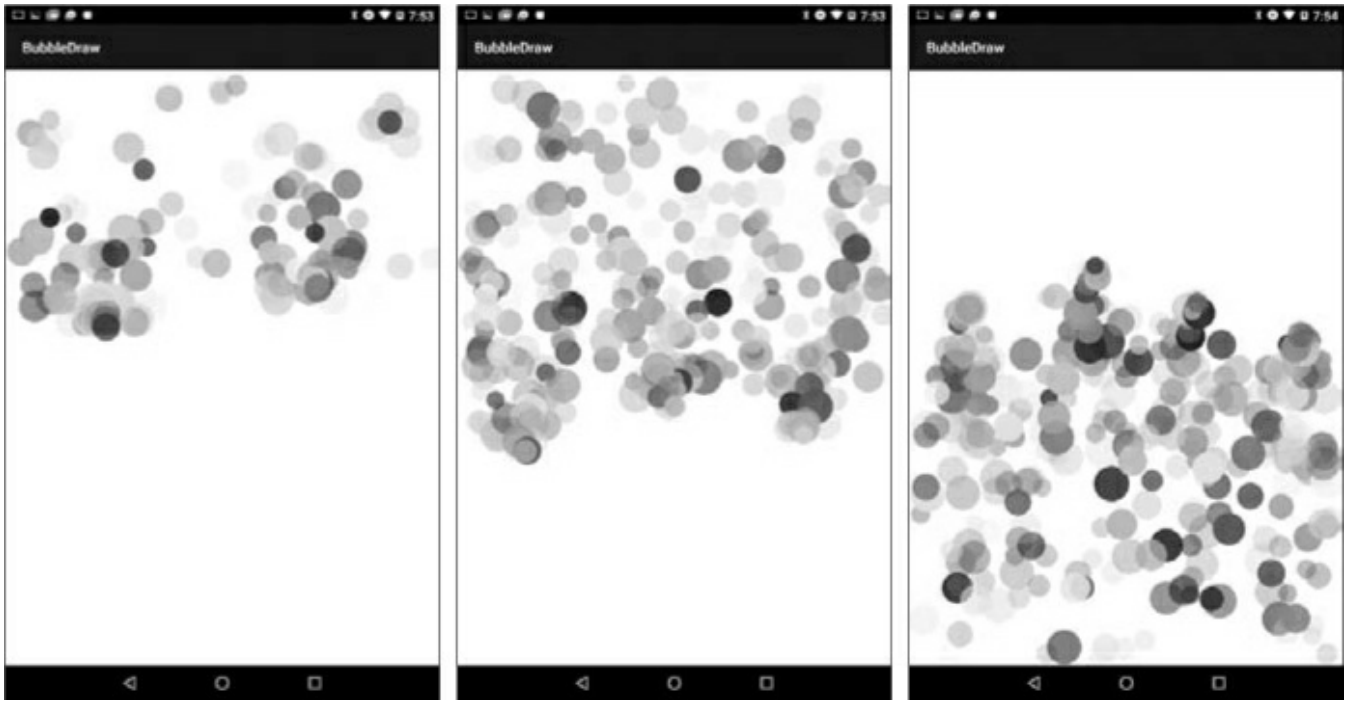


Figure 11-9: Touching the screen of my Android device with two (left), three (center), or four fingers (right) creates multiple streams of colorful, bouncing bubbles.

As you drag your fingers across the screen, bubbles seem to flow from your fingertips, whether you use one finger, two, five, or even ten!

There's one other cool feature we didn't mention yet: to clear the screen, just turn your device sideways (make sure your device's orientation lock is set to Auto-rotate, not Portrait view). The change in orientation forces the app to reinitialize the `BubbleView`, which resets the `bubbleList` and clears the screen. It's a cool effect, and it makes the app feel even more tactile and interactive.

There's just one more customization we'll make to our final BubbleDraw app.

We'll replace the default Android app icon with an icon of our own.

Changing the App Launcher Icon

All our apps so far have used the default Android app launcher icons, which feature Android's friendly green droid. But what if you wanted to use your own, custom icon for the BubbleDraw app—for example, your company's logo or a screenshot from the app like the one shown in Figure 11-10?

In order to give your app its own custom icon, you'll need to create your own *ic_launcher.png* file, paste it into the *app* ▸ *src* ▸ *main* ▸ *res* ▸ *drawable* folder, and then modify the *AndroidManifest.xml* file to use your new icon as the app launcher icon for your app.

Figure 11-10: A custom image cut from a BubbleDraw screenshot

Creating a Custom App Icon

By default, Android names the launcher icon for your app *ic_launcher.png*. If you open the *app* ▸ *src* ▸ *main* ▸ *res* ▸ *mipmap* folder, you'll see several *ic_launcher.png* files of various sizes—stored inside folders labeled *mipmap _mdpi*, *mipmap_xxhdpi*, and so on—that correlate to the various screen sizes of different phones and tablets.

We'll call our new image *ic_launcher.png* as well, just for convenience. Using your favorite image-editing program, create a PNG file with the image you want to use for your app launcher icon. (The site *http://www.gimp.org/* has a great, free image editor, and *https://www.pixlr.com/editor/* is also free over the web.) It's best to use a square image, but Android can also work with files that aren't perfectly square. My image measures 156 × 156 pixels, but anything between 64 × 64 and 256 × 256 should work fine.

NOTE

If you want to use a screenshot of the app like I did, taking a screenshot is easy: press and hold your Android device's sleep/wake button and the volume-down button at the same time until the screen flashes to let you know the image is saved. In your Photos app, find Screenshots, and you should see the image. You can edit the image to be 256 × 256 pixels or smaller by emailing it to yourself so you can access it on your regular computer, or by cropping the image directly on your Android device and then emailing it to yourself.

Save or export your file from the image editor as *ic_launcher.png*. In the next step, we'll copy this image and paste it into the BubbleDraw project in Android Studio.

Adding the Custom Icon to Your App

Once you've created your own *ic_launcher.png* app icon file, open the folder you saved it into and copy the file.

In Android Studio, under the Project Explorer pane for your BubbleDraw app, find the *app* ▸ *res* ▸ *drawable* or *app* ▸ *src* ▸ *main* ▸ *res* ▸ *drawable* folder and paste your new *ic_launcher.png* image into it, as shown in Figure 11-11.

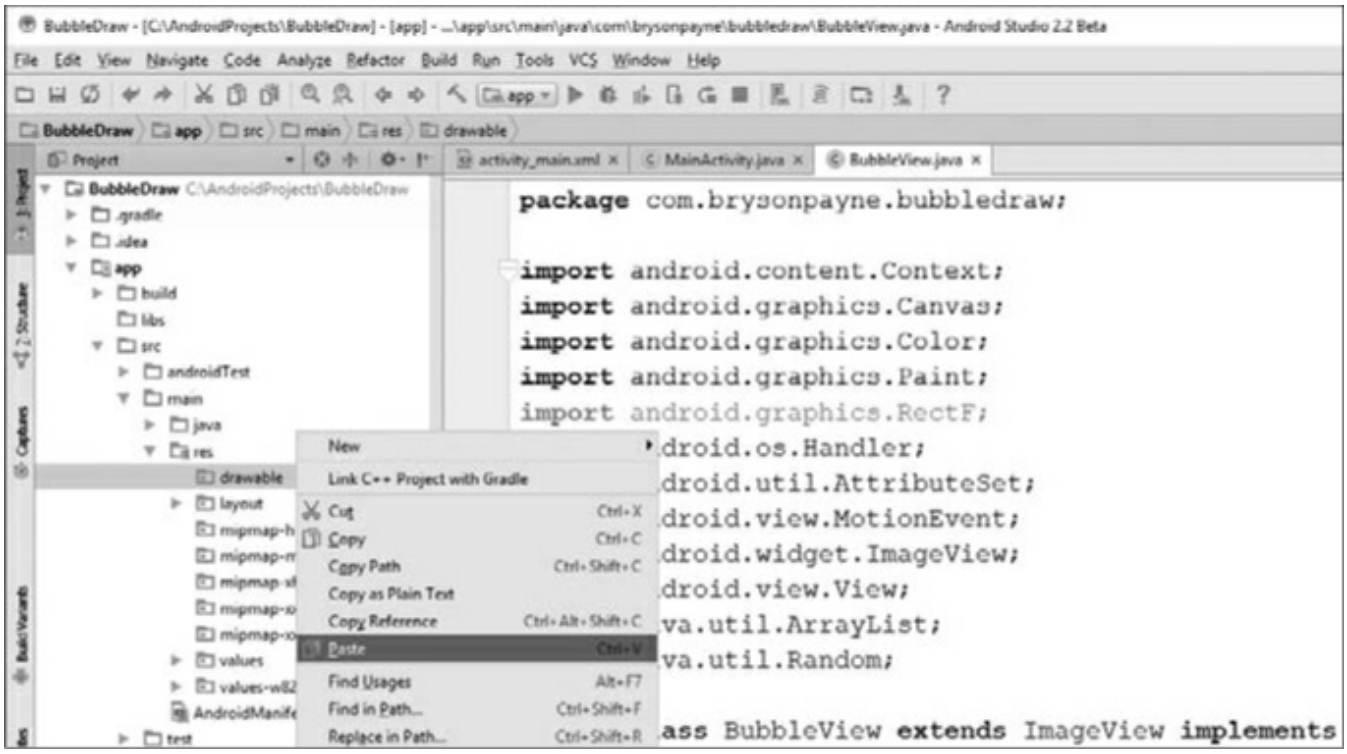


Figure 11-11: Paste your new, custom app icon into the app ▸ src ▸ main ▸ res ▸ drawable *folder.*

You'll see an Android Copy window verifying the location of the file. Click **OK**.

When the icon copies into your *drawable* folder, you'll be able to open the folder and double-click *ic_launcher.png* to preview it in Android Studio.

We now have the new icon we created in the BubbleDraw project's structure, so we can tell Android to use this image as the app icon.

Displaying Your New Icon

For this last step, we'll edit the *AndroidManifest.xml* file for your app. The *AndroidManifest.xml* file describes some of the basic structure, properties, and functionality of your app to the Android operating system.

Under *app* ▸ *src* ▸ *main* (or *app* ▸ *manifests*), open *AndroidManifest.xml*. Near the top of the file, find the entry for `android:icon`, and change it to the new file you just placed in the *drawable* folder:

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
```

We're telling Android to look inside this project's *drawable* folder and find an image file named *ic_launcher*. We leave the *.png* extension off in the manifest.

Now, save the *AndroidManifest.xml* file and press the run button to compile and deploy the app with the new icon. Once the app updates on your emulator or Android device, you'll see the new icon for BubbleDraw, as shown in Figure 11-12.

The custom PNG image we created is now the icon for the app in Android!

Figure 11-12: We've customized the app launcher icon for the BubbleDraw app.

Changing the App Name

You can also customize the app's name as it appears on both the app icon and the title bar inside the app. In your Project Explorer tab on the left, under *app* ▸ *src* ▸ *main* ▸ *res* ▸ *values*, open *strings.xml*. The *strings.xml* file stores the `app_name` string that's used both on the launcher icon, as shown in Figure 11-12, and in the title bar of the app while it's running.

Since BubbleDraw without a space between the words is a bit strange, we'll add a space between the words in the line of XML that defines the `app_name` variable in *strings.xml*:

```
<resources>
    <string name="app_name">Bubble Draw</string>
</resources>
```

Of course, you can make this app name anything you want, like *Your Name*`\'s Bubble Draw App`, but only about 11 or 12 characters will fit below the app icon, so you'd see something like *Your Name*`'s...` on your home screen. You also would need to escape any special characters in the app name, including the single quote, with the backslash character.

Now that you know how, go back and customize your Hi-Lo guessing game and Secret Messages apps with their own icons as well. Keep refining your apps and trying new things—it's the best way to keep learning as you code.

What You Learned

You've come a long way since Chapter 1. After building three complete mobile and desktop apps, you've gained significant computing skills and put those newfound abilities to use. Perhaps most importantly of all, you learned how to improve the apps step-by-step by adding more and more functionality until each app does exactly what you want it to do.

In this chapter, you reinforced several skills and added the following abilities to your coding tool chest:

- Drawing graphics in Android
- Adding a new class to an Android Studio project
- Setting up variables and importing classes
- Building a class constructor from scratch
- Converting Java AWT graphics to Android `Canvas` graphics
- Drawing on an `ImageView` using the `onDraw()` method
- Creating a `Runnable` object to implement threading in Java
- Using a `Handler` to communicate with a separate thread in Java
- Using threading to improve animation efficiency
- Handling multiple touch events and using `MotionEvent` methods to locate touch events onscreen
- Customizing the app launcher icon and app name

Programming Challenges

Try these programming challenge exercises to review and practice what you've learned, as well as to expand your programming skills. Visit the book's website at *https://www.nostarch.com/learnjava/* for sample solutions.

#1: Combining One-Finger and Multitouch Events, v1.0

In this chapter, we learned to handle a single-touch event and multiple-touch events. In this challenge, your task is to tell the difference between a single-touch and a multitouch event. You'll change the logic inside the `onTouch()` method to draw bigger bubbles when one finger is touching the screen and smaller bubbles when more fingers touch the screen.

Remember, you can find out the number of touch events using the `getPointerCount()` on your `MotionEvent` object inside the `onTouch()` method.

For extra practice, code your app so that more fingers will result in smaller and smaller bubbles.

#2: Combining One-Finger and Multitouch Events, v2.0

Once you've mastered Programming Challenge #1, give this one a try. Modify both the `Bubble` class and the `onTouch()` listener to group bubbles when you draw with one finger by giving them all the same `xspeed` and `yspeed` values, but still allow multitouch to blow bubbles in every direction.

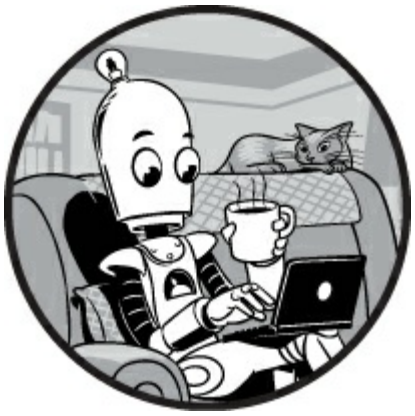
This will require changes to the `onTouch()` listener similar to those in Programming Challenge #1. But, to group the bubbles, you'll need a way to draw some bubbles with fixed speeds and other bubbles with random speeds. The bubbles drawn during a single-touch event should all move at the same fixed speed so that they appear grouped, and bubbles drawn with multitouch should continue to be assigned random speed values.

To accomplish this, you can create a second `Bubble()` constructor. It will be similar to the `Bubble(x, y, size)` constructor that we created for the `Bubble` class originally, but it will have a different number of parameters.

For example, you could create a second `Bubble()` constructor that takes the five parameters `x`, `y`, `size`, `xspeed`, and `yspeed`. Then, you could use that constructor whenever the user touches the screen with just one finger to give all the bubbles the same speed values. You'd use the original constructor whenever the user

touches more than one finger to the screen so the bubbles flow in random directions.

DEBUGGING AND AVOIDING COMMON ERRORS IN JAVA



While working through this book, you've probably made a few typos or mistakes along the way. In this appendix, I'll go over a few common programming pitfalls to avoid. One nice feature of both Eclipse and Android Studio that is helpful when debugging is *syntax coloring*. Class names, functions, variable types, strings, comments, and so on are each colored differently in the IDE based on their syntax.

Syntax coloring can help you spot typos and other problems quickly. For example, if you forget to close a string with a second pair of double quotes, the semicolon at the end of that line will be colored differently from the other semicolons on the screen. As we go through these common errors, try introducing them into one of the apps you've written and look for the warnings given by Eclipse and Android Studio. It's okay if you break an app—you can always go back and fix it, either by using the original code listing in the book or by pressing CTRL-Z (⌘-Z) or clicking **Edit ▸ Undo**.

Spelling and Case

Spelling is important in any programming language, but in Java, case matters, too. For example, if you use a lowercase `s` to make a scanner or string, the text editor in Eclipse will underline the class name in red, and Android Studio will color the text red. It might seem silly, but Java only understands `String`, not

`string`.

The same goes for variable names, such as `playAgain` or `theNumber`. For example, if we accidentally capitalize the `P` and use a lowercase `a`, spelling `Playagain`, Java doesn't understand that we mean the variable `playAgain`. Let's take a look at how each IDE helps us find and fix typographical errors like these, first in Eclipse and then in Android Studio.

Correcting Typos in Eclipse

Eclipse will alert us to an error by underlining the misspelled word in red, and Android will color the text. Figure A-1 shows how Eclipse highlights two errors I've introduced into the Hi-Lo guessing game from Chapter 2.

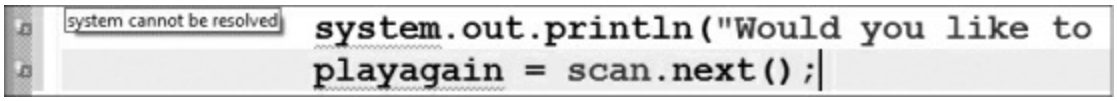


Figure A-1: Eclipse helps us find misspellings and errors in capitalization.

Where I've used a lowercase `s` in `system`, Eclipse underlines the misspelling and adds an error alert along the left border of the screen that you can see by mousing over that line. The Java compiler issues the error `"system cannot be resolved"`. That means Java can't figure out what `system` is supposed to be because it only understands `System` with a capital `S`. Eclipse does the same with `playagain` on the next line—it should be `playAgain`.

Remember that you can use the content assist feature in Eclipse to fix many errors like these. Click one of the misspelled words from Figure A-1, and you'll see a content assist menu like the ones shown in Figure A-2.

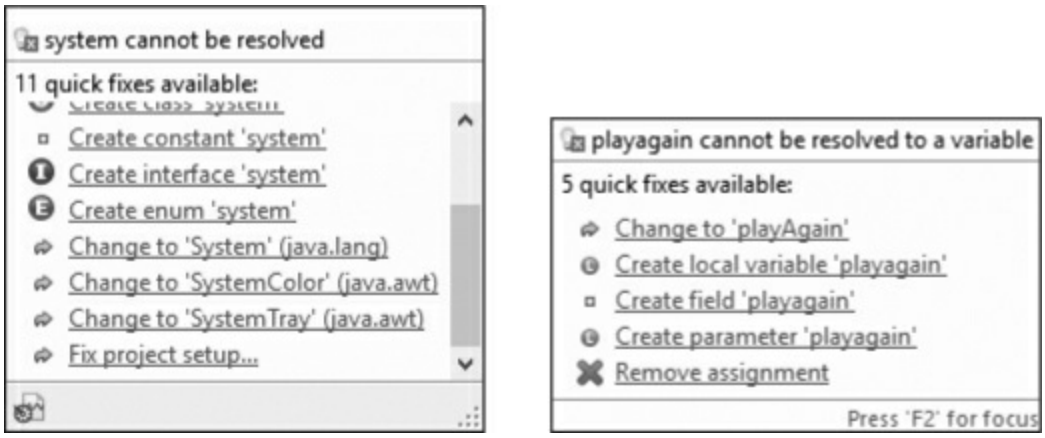


Figure A-2: Eclipse's content assist feature provides information about errors and offers possible fixes.

For the first misspelling, Eclipse not only tells us that it can't resolve or understand `system` but also offers several possible corrections, including, fourth from the bottom in Figure A-2, `Change to 'System' (java.lang)`. The content assist feature doesn't always have the right answer, but in this case, one of the 11 quick fixes offered is the correct capitalization of `System`. For the second misspelling, the first quick fix Eclipse offers is the right answer, `Change to 'playAgain'`. If Eclipse's content assist offers the right fix, click the correct entry, and your spelling or capitalization error will be replaced by the correct code.

Correcting Typos in Android Studio

Android Studio tries to help us find errors by coloring the suspected problem in red. In the example shown in Figure A-3, I've misspelled the keyword `public` by typing `Public` with a capital `P`.

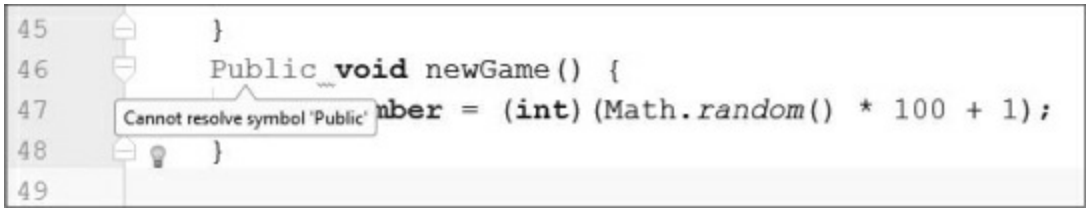


Figure A-3: Android Studio alerts us to an incorrectly capitalized keyword.

Notice that the error is similar to the one we saw in Eclipse earlier: Java is telling us that it can't resolve or understand the symbol `'Public'`. However, Android Studio may not always provide the same level of support in fixing errors, as shown in Figure A-4.

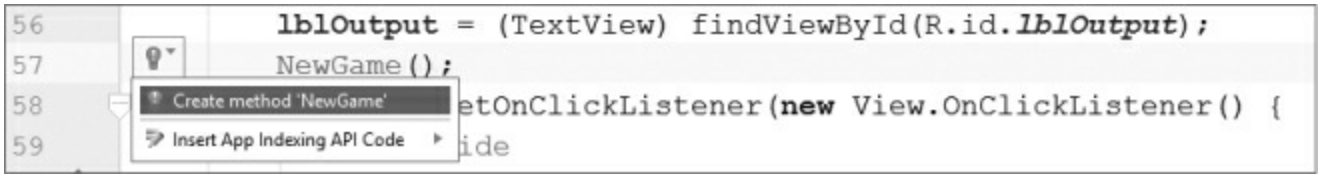


Figure A-4: Pressing ALT-ENTER(or OPTION-ENTER) in Android Studio offers a quick fix, but not quite the right fix for this error.

In Figure A-4, I've incorrectly typed `newGame()` as `NewGame()` with a capital `N`. Android Studio correctly colored the code red to let me know there's an error, but notice what happens when I click the code and press ALT-ENTER to attempt a quick fix. Instead of suggesting `newGame` as the correct spelling, Android Studio offers to create another method with the name `NewGame`. This isn't the right fix for the mistyped method name, so you'll need to find the correct spelling where

the method `public void newGame()` was defined and manually fix the typo.

Both Android Studio and Eclipse will do their best to help you find errors, and they'll often provide possible quick fixes, but knowing that spelling and capitalization are common issues can help you avoid and quickly correct errors as they pop up in your code.

Avoiding Other Common Spelling Errors

One error that is particularly difficult to debug is when the name of an event handler or other overridden method is mistyped. These typos can range from typing `OnCreate()` or `OnDraw()` instead of the correct `onCreate()` and `onDraw()`, to mistyping `public void actionPerformed(ActionEvent e)` inside an `ActionListener` for a button in a Java GUI app, or leaving off the "ed" in the `mousePressed()`, `mouseClicked()`, or `mouseDragged()` event handlers.

We have programming conventions in order to avoid these types of errors. Forgetting to use lowercase to start method or variable names but uppercase to start class names may not necessarily be an error, but it's bad form in Java. If you get into the habit of using Java conventions like these from the beginning, you'll save time and avoid frustration in the future.

Comparison Trouble

Remember that the double equal sign (`==`) is the comparison operator for "is equal to." Don't confuse it with the assignment operator (`=`). For example, to assign the value `5` to a variable named `number`, you would enter:

```
int number = 5; // Assigns the value 5 to number
```

But, to compare the value of the variable, you use the double equal sign:

```
if (number == 5) // If number "is equal to" 5
```

Also, remember that we check strings using the `equals()` method instead of the `==` operator. For example, the condition `if (playAgain == "y")` would always evaluate to `false`. The correct `if` statement would be this:

```
if (playAgain.equals("y"))
```

The `equals()` method of the `String` class checks whether the *contents* of the strings match, which is usually what we want when we compare strings. The `equals()` method is also what we use whenever we compare two objects, as in `if (bubble1.equals(bubble2))` to see whether two `Bubble` variables refer to the same `Bubble` object in the BubbleDraw app, for example.

Grouping Symbols

When coding in Java, it's also important to keep your parentheses, brackets, and braces balanced. We call these *grouping symbols* because they group other programming elements together and because you should never have an opening grouping symbol without a closing grouping symbol. Both Eclipse and Android Studio offer different ways to spot and fix unclosed grouping symbols.

Quick Fixes in Eclipse

Eclipse can help us with grouping symbols in a couple of ways. If you leave out a parenthesis from a condition or function, Eclipse will indicate the error with a red underline beneath the word closest to the missing parenthesis, as shown in Figure A-5.

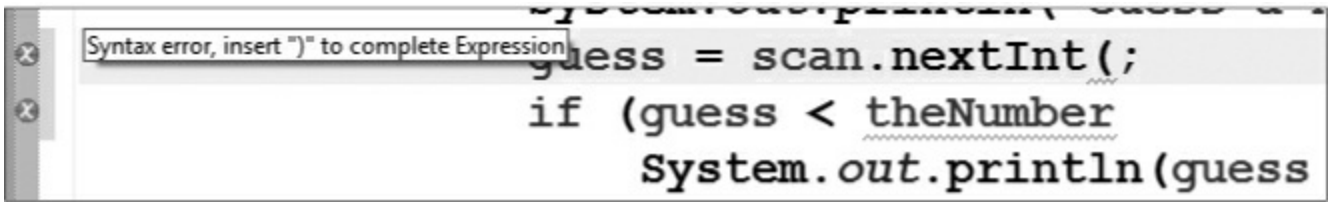


Figure A-5: Eclipse helps find errors with parentheses and sometimes offers a fix.

Again, Eclipse will do its best to help you spot errors while you type. In Figure A-5, I left out closing parentheses at the ends of two lines, which Eclipse marked with a red error symbol. Clicking the red error marker on the left margin of the editor for the first line offers the suggestion to insert a closing parenthesis.

If you place the cursor beside a closing or opening brace, Eclipse will outline the corresponding brace that either opens or closes that block of text. Try it out by clicking the cursor next to a brace in any of your programs.

Also, remember that Eclipse can correct indentation for us if we highlight the

code to indent and press CTRL-I. For short programs, this may just be a nice convenience, but for longer programs that span multiple pages, correct indentation can be very helpful in avoiding errors due to missing or misplaced braces.

Code Completion in Android Studio

Android Studio offers even more help for missing grouping symbols. The code completion feature in Android Studio can automatically fill in both opening and closing parentheses, brackets, or braces.

For example, open the GuessingGame project from Chapter 4 and, inside the `onCreate()` function, find the line that calls a new game:

```
lblOutput = (TextView) findViewById(R.id.lblOutput);
newGame();
btnGuess.setOnClickListener(new View.OnClickListener() {
```

Delete the parentheses and the semicolon at the end of the line:

```
lblOutput = (TextView) findViewById(R.id.lblOutput);
newGame
btnGuess.setOnClickListener(new View.OnClickListener() {
```

Android Studio will color the text red to let you know there's an error, and then you can use code completion to fill in the missing parentheses and semicolon. To complete the statement automatically, position the cursor just after `newGame`, as shown in Figure A-6, and press CTRL-space to perform basic code completion.

Figure A-6: Place the cursor next to a line with missing parentheses (top) and press CTRL-space to finish the statement using Android Studio's code completion (bottom).

Notice that code completion filled in both the missing parentheses and the semicolon to end the line. The CTRL-space combination is just the first of three

code completion shortcuts in Android Studio. The second one, CTRL-SHIFT-space, is called *smart completion* and displays relevant options in a pop-up window. Press the keyboard shortcut twice to expand the list of code completion options. Finally, *statement completion* (CTRL-SHIFT-ENTER on Windows and Linux, ⌘-SHIFT-ENTER on macOS) will add closing parentheses/brackets/braces and a finishing semicolon if needed. Try statement completion by deleting just the closing parenthesis from a method, `if` statement, or `for` loop and pressing CTRL-SHIFT-ENTER. The statement completion feature in Android Studio will often close your grouping symbols and add semicolons or braces wherever they're needed, making your job easier and faster and helping you avoid errors while you code.

Summary

These are far from the only errors you'll run across while programming in Java, but you can see that both Eclipse and Android Studio offer very helpful tools for beginners and experienced professionals alike. You'll get better at spotting and correcting errors the more you code in Java, but your code will probably never be completely error-free. I've been programming for over 30 years, and I still have to debug my code. To avoid, find, and fix errors in your code as you go, it's important to learn good programming practices from the beginning and to take advantage of the support built into professional tools such as Eclipse and Android Studio.

INDEX

Symbols

+ (addition operator), 12
= (assignment operator), 13–15, 245, 276
{ } (braces), 23
+ (concatenation operator), 12–13, 27, 30, 130–131
-- (decrement operator), 130
. (dot operator), 59, 68
" (double quotation marks), 27
== (equal-to operator), 30, 276
// (forward slashes), 23, 28
> (greater-than operator), 30, 60
++ (increment operator), 129
< (less-than operator), 30, 60
% (modulo operator), 139
!= (not-equal-to operator), 29
; (semicolon), 24
~ (tilde), 16
* (wildcard character), 46

A

About screen, 113–114
AbsoluteLayout value, 240
ActionListener class, 63–64, 226
action listeners, 46–47, 68, 234
actionPerformed() method, 63, 154–156, 227–228, 234
addActionListener() method, 63
addMouseListener() method, 213
addition operator (+), 12

`AlertDialog` class, 113–114
alert dialog pop-up, 113–114
Android devices
    connecting to computer, 101
    Developer mode, 100–101
    preparing to run apps, 100
    testing apps on, 100–102
Android emulator
    BubbleDraw app, 260–261
    Hi-Lo guessing game, 94–99
    Secret Messages app, 188–189
Android package files (APK), 101–102
Android Studio, 1. *See also* BubbleDraw app (Android mobile version); Hi-Lo guessing game (Android mobile version); Secret Messages app (Android mobile version)
    activity selection, 78–79
    Android emulator, 94–99
    code completion, 91–92, 278
    correcting spelling errors, 275
    default project view, 80
    Design tab, 80–81
    grouping symbols, 277–278
    importing classes automatically, 86
    installing, 8–9
    SDK or API levels, 78
    starting projects, 76–80
    Target Android Devices window, 78
    updating, 176
    widgets, 81
animation, 225–229. *See also* BubbleDraw app (Android mobile version); BubbleDraw app (GUI version)
    adding timer, 226

frame rate, 225, 241, 253
preparing, 227–228
setting timer, 226–227
starting timer, 229
threading, 249
anonymous inner classes, 63, 91, 155, 167, 184, 210
APK (Android package files), 101–102
application program interface (API) levels, 78
`apply()` method, 119
arguments, 155
arithmetic operators (`+`, `-`, `*`, `/`), 12
`ArrayList` data structure, 205–210, 253
assignment operator (`=`), 13–15, 245, 276
attributes, 26

B

binary search, 40
bin folder, 1, 11–12, 141–143
Boolean expressions, 65
braces (`{ }`), 23
`break` statement, 113
`Bubble` class, 200–205, 255–257
attributes for, 201
defining bubbles, 200–202
methods for, 202–205
modifying, 255–257
BubbleDraw app (Android mobile version)
adding animation variables, 252–253
adding `testBubbles()` method, 258–259
`BubbleView()` constructor, 254
custom app launcher icon, 268–270
adding, 268–269

changing app name, 270
creating, 268
displaying, 269–270
drawing with `onDraw()` method, 257–258
handling `OnTouchListener` events, 259–260, 264-265
modifying `Bubble` class, 255–257
running in Android emulator, 260–261
setting up project, 250–252
testing, 258–261, 266–267
threading and multitasking, 261–264
touch event handler code, 264–265
combining single-pointer and multitouch events, 271–272
multitouch, 266–267
BubbleDraw app (desktop version)
building frame, 199
creating `Bubble` class, 200–205
attributes for, 201
defining bubbles, 200–202
methods for, 202–205
creating project files, 198–199
mouse events, 210–217
clicks, 212–214
creating reusable event listener, 211–212
drags, 214–215
wheel events, 215–217
pixelated effect, 219–220
`size` variable limits, 218
storing bubbles in `ArrayList`, 205–210
adding constructor, 206–207
adding method, 207–208
testing `BubblePanel` class, 208–210
BubbleDraw app (GUI version)

adding transparency, 224–225
animation, 225–229
adding timer, 226
frame rate, 225, 241
preparing, 227–228
setting timer, 226–227
starting timer, 229
building GUI, 232–235
event handlers, 234–235
setting up panel and buttons, 232–233
collision detection, 235–239
hard bounces, 238–239
soft bounces, 236–238
copying project and renaming Java file, 222–223
fixed speed, 244–245
pixelated effect, 245–247
preventing stationary bubbles, 244
random speed and direction, 229–232
sharing app as runnable JAR file, 243
speed control slider, 240–243
customizing, 241
event handler, 242–243
`BubbleDraw` class, 199
`BubbleListener` class, 212–214, 227
`Bubble()` method, 202–204
`BubblePanel` class, 199, 201, 203, 206–209, 214–215
`BubbleView` class, 251–255
`BubbleView()` constructor, 254
`Button` component, 82, 84, 87, 180
buttons
adding, 82

adding event listeners, 62–64, 91–94
adding labeled, 46–47, 54
Clear button, 232–235
Click me! button, 46–47
Encode/Decode button, 152, 154–156, 159–164, 167, 179–180, 182–186
floating action button (fab), 84–85, 190–193
Guess! button, 54–55, 62–63, 66–69, 82, 93, 99
Move Up ^ button, 172–173, 196
OK button, 114
Pause/Start button, 232–235
Play Again button, 66, 73–74
showing and hiding, 73–74

C

Caesar cipher, 125–126
camel case, 14, 21
`Canvas` class, 256
case sensitivity, 14, 21, 274
casts (type casts), 24, 132
chained assignment, 245
`charAt()` method, 130–132
`char` data type, 132
`checkGuess()` method, 58–59, 65, 68–69, 73, 88–89, 106, 116, 120
child classes, 49
classes, 21, 86–87
anonymous inner classes, 63, 91, 155, 167, 184, 210
creating, 21–23, 200–204
extending, 49
importing, 25
public, 23
superclasses, 48
Clear button, 232–235

Click me! button, 46–47
`close()` method, 37, 135
code completion, 91–92, 278
code recommenders, 59, 274–275
collision detection, 235–239
    hard bounces, 238–239
    soft bounces, 236–238
Color Chooser dialog (in Eclipse), 160, 165
`Color` class, 201, 203, 224, 255–256
command line apps, running without Eclipse, 141–144
    opening command line window, 142–144
    workspace folders, 141–142
comments, 23, 28
comparison operators
    equal to (==), 30, 276
    greater than (>), 30, 60
    less than (<), 30, 60
    not equal to (!=), 29
compiler directives, 91
concatenation operator (+), 12–13, 27, 30, 130–131
conditional expressions, 30
conditions, 28
confirm dialogs, 163
content assist, 59, 274–275
content panes, 52

D

data structures, 205
debugging, 28
    case sensitivity, 274
    code completion, 278
    commenting out, 28

grouping symbols, 276–278
spelling errors, 274–276
statement completion, 278
syntax coloring, 273
USB debugging, 100–102, 188, 267
decrement operator (--), 130
`defaultCloseOperation` property, 51–52
`Dimension` class, 64–65
`dismiss()` method, 114
dot notation, 26
dot operator (.), 59, 68
double quotation marks ("), 27
`double` variable type (double-precision floating point), 14
`do-while` loops, 33–35
`draw()` method, 204–205, 247, 256
`drawOval()` method, 250, 256–257
`drawRect()` method, 204

E

Eclipse, 1. *See also* BubbleDraw app (GUI version); Hi-Lo guessing game (GUI version); Secret Messages app (GUI version)
automatic indentation clean-up, 35
content assist and code recommenders, 59, 274–275
correcting spelling errors, 274–275
creating classes, 21–23
creating new Java project, 20–21
customizing appearance of, 7–8
grouping symbols, 277
installing, 3–4
perspectives, 21
saving code, 25
setting up

default workspace storage location, 4–5
installing WindowBuilder Editor, 6–7
modifying to run with Java 9, 5
setting up GUI projects, 48–49
themes, 7
WindowBuilder Editor, 50
Design tab, 50
installing, 6–7
Palette pane, 52–54
Properties pane, 51–52
Source tab, 56
workspace folders, 4–5, 141–142
`/edit` command, 16
editor action, 103
`EditText` component, 82, 84, 87, 102, 107, 178, 185
`else-if` statements, 138–139
`else` statements, 31
encapsulation, 201
Encode/Decode button, 152, 154–156, 159–164, 167, 179–180, 182–186
`encode()` method, 151–155, 182–185
`equalsIgnoreCase()` method, 34
`equals()` method, 34, 276
equal-to operator (`==`), 30, 276
escape character sequence (`\n`), 107, 178
event listeners, 62, 117. *See also* mouse events; touch events
for buttons, 62–64, 91–94
creating reusable, 211–212
for ENTER key, 103–104
exception handlers, 70–72
exceptions, 70
`/exit` command, 16

expressions, evaluating in JShell, 12–13
extending classes, 49

F

fab (floating action button) icon, 84–85, 190–193
`fillOval()` method, 204–205
`fillRect()` method, 220, 247
`finally` block, 71–72
`findViewById()` method, 87
floating action button (fab) icon, 84–85, 190–193
floating-point numbers, 14
`FlowLayout` value, 240
focus, 69
Font Chooser dialog (in Eclipse), 159
`for-each` loops, 228, 263
`for` loops, 129–134
forward slashes (`//`), 23, 28

G

game loops (`do-while` loops), 33–35
`getHeight()` method, 236–238
`getInt()` method, 120
`getMenuInflator()` method, 111
`getPointerCount()` method, 266, 271
`getProgress()` method, 186
`getSource()` method, 235
`getStringExtra()` method, 194
`getText()` method, 59, 155, 185
`getUnitsToScroll()` method, 216
`getValue()` method, 167–168, 243
`getWidth()` method, 236–238
`Graphics` class, 204

greater-than operator (>), 30, 60
grouping symbols, 276–278
Guess! button, 54–55, 62–63, 66–69, 82, 93, 99

H

`Handler` class, 253, 262
`"Hello, Java!"` test code, 9
`/help` command, 16
`HiLo` class, 21–23
Hi-Lo guessing game (Android mobile version)
    adding method to begin new game, 90–91
    adding method to check player's guess, 88–89
    building GUI layout, 80–83
    changing guessing range, 114–117
        adding variable, 115
        dialog to allow user to select range, 116–117
        modifying behavior of app to use variable, 115–116
    changing number of tries, 122–123
    event listeners, 91–94
    options menu, 109–114
        adding items to XML file, 110–111
        alert dialog pop-up for About screen, 113–114
        displaying, 111–112
        responding to user selections, 112–113
    ratio of wins to losses, 123
    running in Android emulator, 94–99
    running on real Android device, 100–102
    setting up project, 76–80
    storing game stats, 120–122
    storing user preferences, 118–120
    Toast messages, 106
    UX improvements, 102–105

aligning text field, 104
centering input in text field, 103
event listener for ENTER key, 103–104
visual improvements, 107
wiring up, 83–88
Hi-Lo guessing game (command line version)
completed code for, 37–38
counting tries, 40–41
creating `HiLo` class, 21–23
creating new Java project, 20–21
generating random numbers, 23–25
getting keyboard input, 25–27
increasing range of numbers, 40
loops, 28–35
`if` statements, 30–32
play again feature, 33–35
making program print output, 27–28
overview of, 19–20
planning basic version of, 20
prompting for user input, 26
testing, 36–37
thank-you for playing, 36–37
Hi-Lo guessing game (GUI version)
adding method to check player's guess, 58
converting strings to numbers, 60–61
getting text from text field, 59–60
designing GUI, 50–58
aligning elements, 54–55
customizing components, 52–54
setting properties, 51–52
wiring up, 55–58
event listeners, 62–64

handling bad input, 69–72
new game (picking new random number), 61–62
play again feature, 66–67, 73–74
setting up GUI window, 64–65
setting up project, 48–49
showing and hiding Play Again button, 73–74
showing number of tries, 73
testing, 66
UX improvements, 67–70
automatically removing old guesses, 68–69
pressing ENTER to guess, 68
`/history` command, 16

I

`if-else` statements, 31, 135
`if` statements, 30–32, 134
image editors, 268
`ImageView` class, 251, 253
`/imports` command, 16
increment operator (`++`), 129
indentation, 30, 35, 277
indexes, 129
`inflate()` method, 111
inheritance, 48
input dialogs, 163
`Integer` class, 60
`int` variable type (integer), 13
`invalidate()` method, 259, 263
`isSpace()` method, 135
`<item>` tag, 111
iteration, 128

J

JAR (Java archive) files, 169–171, 243
Java, 1
    as first programming language, xx
    modifying Eclipse to run with Java 9, 5
    as object-oriented programming language, 21
    printing output in, 15
    testing setup using JShell, 9–17
Java archive (JAR) files, 169–171, 243
Java Development Kit (JDK), 2–3, 10
Java Perspective (in Eclipse), 21
Java Runtime Environment (JRE) software, 1–2, 141
Java Virtual Machine (JVM) software, 1–2, 141
`JButton` component, 46, 52, 54, 63, 74, 151, 232
`JCheckBox` component, 52
JDK (Java Development Kit), 2–3, 10
`JFrame` class, 44, 48–49, 199, 223
`JLabel` component, 52–54, 57, 61, 74, 151, 240
`JOptionPane` component, 164
`JPanel` component, 46, 198–199, 207–208, 232, 240
JRE (Java Runtime Environment) software, 1–2, 141
`JScrollPane` component, 173–174
JShell, 9
    commands, 16–17
    creating GUIs in, 44–47
    declaring Java variables in, 13–15
    evaluating Java expressions in, 12–13
    "`Hello, Java!`" test code, 9
    opening code in, 16
    printing output, 15
    running, 9–11
        from command line, 10–11

confirming installed version of JDK, 10
from shortcut, 11–12
saving code created in, 16
starting new code snippets, 17
testing Java setup, 9–17
JShell Edit Pad, 46–47
`JSlider` component, 164–167
`JTextArea` component, 150–152
scroll bars for, 173–174
wrapping text, 162
`JTextField` component, 52–53, 56–57, 59–60, 68–69, 151–152
JVM (Java Virtual Machine) software, 1–2, 141

K

key/value pairs, 118

L

labels
adding, 52–54, 81–82
alignment, 53
editing, 53, 81–82
font, 53
naming for coding, 56
`length()` method, 129–131
less-than operator (<), 30, 60
`lineWrap` property, 162
`/list` command, 16
logarithms, 123
`log()` method, 123
loops, 28–35, 144–145
`do-while` loops, 33–35
`for-each` loops, 228, 263

`for` loops, 129–134
nested, 38
`while` loops, 28–30

M

`MadLibs` class, 41
MadLibs game
Android mobile version, 107
command line version, 41
GUI version, 74
`main()` method
BubbleDraw app
desktop version, 210
GUI version, 222–223
Hi-Lo guessing game
command line version, 22–23, 25, 29, 35
GUI version, 48, 64–65
Secret Messages app
command line version, 127–128, 135–136
GUI version, 148, 156–157
`majorTickSpacing` property, 165, 241
`makeText()` method, 106
manifest file, 193
`Math` class, 23
math expressions, 12
`MenuInflator` class, 111
`<menu>` tag, 110–111
message dialogs, 163–164
methods, 22, 200–210
designing, 202–205, 207–208
displaying available methods with content assist, 68–69
method signature, 207

minorTickSpacing property, 165, 241
modulo operator (%), 139
MouseAdapter class, 212
mouseClicked() method, 212
mouseDragged() method, 214–215
MouseEvent class, 212–213
mouse events, 210–217
    clicks, 212–214
    creating reusable event listener, 211–212
    drags, 214–215
    wheel events, 215–217
mouseMoved() method, 214
mousePressed() method, 212–214
mouseReleased() method, 212
MouseWheelEvent class, 215–216
mouseWheelMoved() method, 216, 218
Move Up ^ button, 172–173, 196
multitasking, 249, 261–264
multitouch, 249, 266–267

N

\n (escape character sequence), 107, 178
nested loops, 38
newGame() method, 61–62, 64, 66–67, 90, 113–115, 120
New Java Class dialog (in Eclipse), 22–23, 49, 148
New Java Project dialog (in Eclipse), 20–21
newline character, 107, 178
nextInt() method, 25, 27–28, 203–204, 224, 231
nextLine() method, 25, 128
not-equal-to operator (!=), 29
numeric variables, 13–14

O

object-oriented programming, 21, 198
objects, 21
OK button, 114
`OnClickListener` event handler class, 91–92, 184
`onClick()` method, 93, 114, 184–185, 191
`onCreate()` method, 87, 90–91, 115, 120, 182, 191
`onCreateOptionsMenu()` method, 111
`onDraw()` method, 257–258, 263
online resources, xxi–xxii
`onOptionsItemSelected()` method, 112, 116, 118–119, 121
`onProgressChanged()` method, 188
`OnSeekBarChangeListener` event handler, 187–188
`onStartTrackingTouch()` method, 188
`onStopTrackingTouch()` method, 188
`OnTouchListener` interface, 251–252, 259–260
`onTouch()` method, 259–260, 264–265, 271
`/open` command, 16
operators
    arithmetic (`+`, `-`, `*`, `/`), 12
    assignment (`=`), 13–15, 245, 276
    comparison
        equal to (`==`), 30, 276
        greater than (`>`), 30, 60
        less than (`<`), 30, 60
        not equal to (`!=`), 29
    concatenation, 12–13, 27, 30, 130–131
    decrement (`--`), 130
    dot (`.`), 59, 68
    increment (`++`), 129
    modulo (`%`), 139
options menu, adding in Android, 109–114

adding items to XML file, 110–111
alert dialog pop-up for About screen, 113–114
displaying, 111–112
responding to user selections, 112–113

P

package declarations, 85–86
`Paint` class, 253
`paintComponent()` method, 207
parameters, 153
parent classes (superclasses), 48
`parseInt()` method, 155, 185
parsing, 60
Pause/Start button, 232–235
persistent information, storing, 118–121
pixelated effect, 219–220, 245–247
Play Again button, 66, 73–74
pointers, 266
`postDelayed()` method, 263
`println()` method, 15, 26–27
`private` modifier, 57
procedural programming, 198
`public` modifier, 58

R

`Random` class, 203
`random()` method, 24, 203
range settings dialog, 116–117
refactoring, 223
`RelativeLayout` value, 255
`repaint()` method, 209, 213, 215, 228, 234
`requestFocus()` method, 164

`/reset` command, 16
resource leaks, 37, 135
`return` statement, 113
`run()` method, 262
`Runnable` interface, 262

S

`/save` command, 16
`Scanner` class, 25, 37
scrolling
    different operating systems and, 216–217
    mouse wheel events, 215–217
    scroll bars, 173–174
SDK (software development kit) levels, 78
Secret Messages app (Android mobile version)
    connecting Encode button to `encode()` method, 182–185
    customizing fab icon, 190–193
    designing GUI, 177–181
    Move Up ^ button, 196
    receiving secret messages from other apps, 193–195
    running in Android emulator, 188–189
    running on real Android device, 188–190
    `SeekBar` component, 187–188, 196
    setting up project, 176–177
    testing, 185–187
    wiring up, 182–188
Secret Messages app (command line version)
    adding loops to, 144–145
    Caesar cipher, 125–126
    character values, 132–133
    closing all input/output resources, 135–136
    custom key values, 137–138

encoding digits, 138–141
encoding only letters, 133–135
full version of, 140–141
languages other than English, 135
reversing and encoding, 145
setting up project, 126–131
asking user for input, 128
creating project in Eclipse, 127
message reverser, 128–131
strings, 128–131
`try-catch` blocks, 145
Secret Messages app (GUI version)
building `main()` method, 156–157
creating `encode()` method, 151–154
designing GUI, 148–151
event handling, 154–156
full source code, 157–159
handling bad input, 156, 162–164
improving GUI, 159–169
line and word wrap, 161–162
Move Up ^ button, 172–173
numeric slider
adding, 164–166
code cracking with, 166, 168–169
moving when value in text field changes, 174
scroll bars, 173–174
setting up project, 148
sharing as runnable JAR file, 169–171
wiring up, 148–151
`SecretMessagesGUI` class, 152
`SeekBar` component, 179–180, 187–188, 196
`selectAll()` method, 69, 164

Select Deployment Target dialog (in Android Studio), 98, 198
semicolon (;), 24
`setBackground()` method, 207
`setButton()` method, 114
`setColor()` method, 204
`setItems()` method, 117
`setOnClickListener()` method, 183–184, 191
`setOnSeekBarChangeListener()` method, 187
`setOnTouchListener()` method, 265
`setSize()` method, 64
`setText()` method, 61, 107, 156, 167
`setVisible()` method, 45, 65
shared preferences, 118–121
`showMessageDialog()` method, 163
`show()` method, 106
sliders
    numeric
        adding, 164–166
        code cracking with, 166, 168–169
        moving when value in text field changes, 174
    speed control, 240–243
        customizing, 241
        event handler, 242–243
smart completion, 278
Snackbar pop-up messages, 191
software development kit (SDK) levels, 78
spelling errors, 274–276
src (source code) folder, 22, 48, 127, 141, 148
`start()` method, 229
`stateChanged()` method, 166–167, 242
statement completion, 278

`stop()` method, 229
`storeRange()` method, 119
strings, 12
`String` variable type, 14–15, 33
stubs, 22
superclasses (parent classes), 48
`super()` method, 253
`switch` statement, 113
syntax coloring, 273
`System` class, 26
`System.in` object, 25
`System.out` object, 26

T

temporary variables, 12
Terminal, 142
`testBubbles()` method, 208–210, 213, 258–259, 265
`Text` class, 88
text fields
    adding, 82
    aligning, 104
    centering input in, 103
    declaring at top level of class, 56–57
    getting text from, 59
    sizing, 83
    wiring up, 55–56, 83–84
`TextView` component, 81–82, 84, 87, 104, 107, 114–116, 177
`Thread` class, 262
threading, 249, 261–264
threads, 253
tilde (~), 16
`Timer` class, 225–227

`title` property, 51
Toast messages, 106
`toString()` method, 88, 185
touch events, 251–252, 259–260, 264–267, 271–272
transparency, 224–225
`try-catch` statements, 71–72, 145, 156, 162–163
type casts (casts), 24, 132

U

undo, 55
Unicode, 132
`update()` method, 228, 237–238, 255
USB debugging in Android, 100–101
user experience (UX), 67

V

variables
    declaring and initializing, 26
    declaring at top level of class, 56–58
    declaring in JShell, 13–15
    temporary, 12
`/vars` command, 16
`void` modifier, 58

W

what-you-see-is-what-you-get (WYSIWYG) interfaces, 50
`while` loops, 28–30
wildcard character (*), 46
WindowBuilder Editor (in Eclipse), 50–52
    Design tab, 50
    installing, 6–7
    Palette pane, 52–54
    Properties pane, 51–52

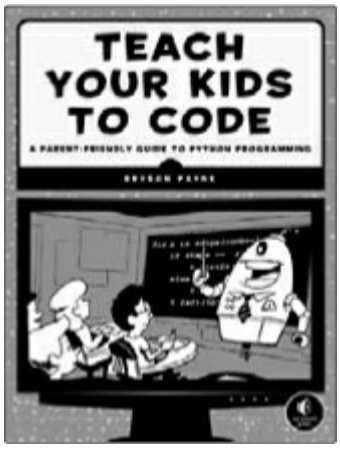
Source tab, 56
windows
close operation, 51–52
content panes, 52
creating, 44–45
sizing, 64
titles, 51
`wrapStyleWord` property, 162
WYSIWYG (what-you-see-is-what-you-get) interfaces, 50

Learn Java the Easy Way is set in New Baskerville, Futura, Dogma, and TheSansMono Condensed.

RESOURCES

Visit *https://www.nostarch.com/learnjava/* for resources, errata, and more information.

More no-nonsense books from **NO STARCH PRESS**



TEACH YOUR KIDS TO CODE
A Parent-Friendly Guide to Python Programming
by BRYSON PAYNE
APRIL 2015, 336 PP., $29.95
ISBN 978-1-59327-614-0
full color

CODING IPHONE APPS FOR KIDS

A Playful Introduction to Swift

by GLORIA WINQUIST *and*
MATT MCCARTHY
JUNE 2017, 336 PP., $29.95
ISBN 978-1-59327-756-7
full color

PYTHON CRASH COURSE

A Hands-On, Project-Based Introduction to Programming

by ERIC MATTHES
NOVEMBER 2015, 560 PP., $39.95
ISBN 978-1-59327-603-4

AUTOMATE THE BORING STUFF WITH PYTHON

Practical Programming for Total Beginners

by AL SWEIGART

APRIL 2015, 504 PP., $29.95

ISBN 978-1-59327-599-0

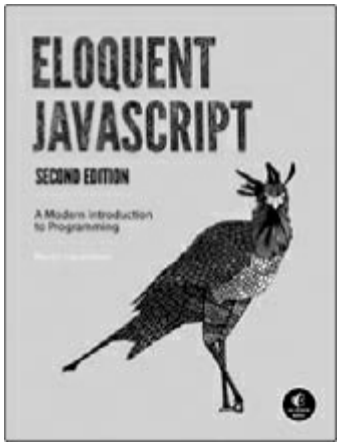
THE LINUX COMMAND LINE

A Complete Introduction

by WILLIAM E. SHOTTS, JR.

JANUARY 2012, 480 PP., $39.95

ISBN 978-1-59327-389-7



ELOQUENT JAVASCRIPT, 2ND EDITION
A Modern Introduction to Programming

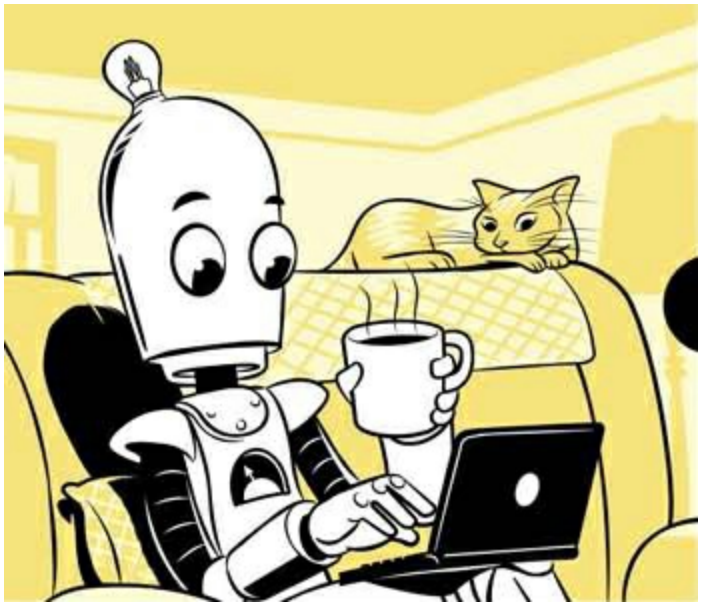
by MARIJN HAVERBEKE
DECEMBER 2014, 472 PP., $39.95
ISBN 978-1-59327-584-6

PHONE:
1.800.420.7240 OR
1.415.863.9900

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

JAVA WITHOUT THE HEADACHE



Java is the world's most popular programming language, but it's known for having a steep learning curve. *Learn Java the Easy Way* takes the chore out of learning Java with hands-on pro|ects that will get you building real, functioning apps right away.

You'll start by familiarizing yourself with JShell, Java's interactive command line shell that allows programmers to run single lines of code and get immediate feedback. Then, you'll create a guessing game, a secret message encoder, and a multitouch bubble-drawing app for both desktop and mobile devices using Eclipse, an industrystandard IDE, and Android Studio, the development environment for making Android apps.

As you build these apps, you'll learn how to:

- Perform calculations, manipulate text strings, and generate random colors
- Use conditions, loops, and methods to make your programs responsive and concise
- Create functions to reuse code and save time
- Build graphical user interface (GUI) elements, including buttons, menus, pop-

ups, and sliders

- Take advantage of Eclipse and Android Studio features to debug your code and find, fix, and prevent common mistakes

If you've been thinking about learning Java, *Learn Java the Easy Way* will bring you up to speed in no time.

ABOUT THE AUTHORS

Bryson Payne is a professor of computer science at the University of North Georgia, where he has taught for more than 17 years. He previously taught middleschool math and programming and continues to work extensively with K—12 schools to promote technology education. He is the author of *Teach Your Kids lo Code*, also from No Starch Press.

COVERS JAYA 8 AND 9

THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com